An Object-Oriented Language for Software Reuse and Distribution

Colin Atkinson

February 1990

Department of Computing,
Imperial College of Science, Technology and Medicine,
University of London,
London SW7 2BZ.

A thesis submitted for the degree of
Doctor of Philosophy of the University of London.

Abstract

There are currently two important challenges facing the developers of large-scale, embedded software systems: one is to increase the extent to which existing software components are reused in the development of new applications, and the other is to take advantage of the prospects for dynamic reconfiguration and fault tolerance made possible by the advent of cheap and powerful distributed processing systems. Until now, however, there has been no single, coherent framework for addressing both these issues. Ada, which was conceived as a standard language for embedded systems development has been found to be lacking in its support for both reuse and distribution, whereas specialised languages, by definition, focus only on one aspect. This is particularly true of object-oriented languages which provide some of the most powerful reuse mechanisms, but which have not previously been extended in a satisfactory way to the realms of concurrency and distribution.

This thesis describes a language, called DRAGOON, which provides a uniform and coherent vehicle for supporting reuse, distribution and reconfiguration in large-scale, embedded systems. The language is primarily "object-oriented", providing the powerful reuse mechanisms of inheritance, polymorphism and dynamic binding, but is also highly "Ada-oriented" in that it incorporates many Ada features for "programming in the small", and can be translated into standard Ada for execution in a natural and straightforward way.

In order to support the development of concurrent and distributed systems the language incorporates two new extensions to the inheritance mechanism of conventional object-oriented languages. In concurrent systems, the synchronisation conditions ruling the relative execution of methods are superimposed on purely "sequential" classes by the inheritance of special predefined "behavioural classes". In distributed systems, classes are endowed with the quality of executability, enabling instances to execute on independent network machines, by inheritance of suitable "execution support classes".

By combining the advantages of the object-oriented and the Ada approaches to software engineering, and extending these into the domains of concurrency and distribution, DRAGOON provides a powerful, unified framework for the development of embedded and large-scale software systems.

Statement of Contribution

The language, DRAGOON, upon which this thesis is based, was developed jointly by three partners of the DRAGON project - TXT S.p.A.(Milan), GSI-Tecsi (Paris) and Imperial College - which is partially funded under the Esprit programme of the Commission of the European Communities. In a development effort that has involved a high degree of collaboration and teamwork it is inherently difficult to provide a precise delineation of each participant's contribution. However, the author would claim to have made significant individual contributions to the DRAGOON development effort in the following areas.

The author's most important single contribution was in the design and specification of the translation into Ada (chapter 7). This lead to his playing an important role in determining the impact of the translation strategy on the form and semantics of all parts of the language. The "Supermarket Control Systems" (section 1.4) was conceived and developed by the author to provide a coherent illustration of the features of DRAGOON, as was all the review material in the thesis, particularly chapter 2, and the introductions to chapter 4, 5 and 6. The views expressed are consequently those of author, not necessarily those of the consortium.

Considering in more detail the contents of chapters 3 to 6, which describe the main features of the language, the author was responsible for :-

- motivating the "mixed" or "hybrid" model of DRAGOON, particularly the use of stateless "template" packages (section 3.1.1),
- providing the rationale for the DRAGOON typing scheme (section 3.5), and identifying the need for type "conversion" facilities (section 3.5.6),
- defining the genericity model described in section 4.4, and relating it to the parametric polymorphism facilities provides by other languages,
- defining the the evaluation semantics of guarded permissions (section 5.6.2),
- proposing the "virtual node" approach to distribution and its realisation within the DRAGOON object oriented framework (section 6.3),
- suggesting the notion of "execution support" and "executable" classes (section 6.5),
- clarifying the DRAGOON execution and configuration model (section 6.8).

At the time of writing there are several detailed aspects of DRAGOON which have not yet been fully finalised. Consequently, the language described in this thesis may not, in its entirity, represent the eventual language standard.

Acknowledgement

I would like to take this opportunity to extend my thanks to all those people who have helped me to produce this work.

My major thanks must go to my supervisor, Professor Stephen Goldsack, who has been a valued friend and colleague over the past four years and to whom I am indebted for making my participation in the DIADEM and DRAGON projects possible. My understanding of the technical material in this thesis has been greatly enhanced through many fruitful hours of discussion with him.

The overall credit for the success of the DRAGON project as a whole, and for the DRAGOON language in particular, must go to Andrea Di Maio, who as project manager has not only made significant technical contributions to DRAGOON, but has lead the DRAGON team with great skill and dedication. It has been a pleasure to work with him on both the DIADEM and DRAGON projects.

This takes nothing away from the contributions made by the other participants in DRAGOON's development - Rami Bayan, Catherine Destombes and Francois Kaag of Tecsi, Cinzia Cradigno and Stefano Genolini of TXT, Stefano Crespi Reghizzi of TXT and the Politecnico di Milano, and Stephen Goldsack of Imperial College - without all of whose efforts DRAGOON would have been significantly the poorer. Rami Bayan, in particular, deserves a special mention for his important influence on the language. Although no clear demarcation of contributions to the language is possible, TXT deserve the credit for the original ideas concerning the concurrency model of the language, while Tecsi have had a major influence on the distribution model.

I am extremely grateful to those colleagues on the DRAGON project, and at Imperial College, who have read and commented on earlier drafts of this thesis. They are Andrew Davison, Guido Galli-De Paratesi, Jeff Magee, Jeremy Pitt, Diana Protic and Ian Sommerville.

Finally no acknowledgement would be complete without thanking my family - my mother, father and sister, for their support and encouragement throughout my studies. I dedicate this work to them.

“ This which I have written may be sufficient to satisfy any reasonable man that there are winged serpents and dragons in the world. ”

- from “The History of Four Footed Beasts and Serpents”,
by Edward Topsel.
(Published in London in 1658).

Contents

Abstract 2

Statement of Contribution 3

Acknowledgement 4

Contents 7

1 Introduction 11
- 1.1 The Role of Ada . . . 12
- 1.2 Object-Oriented Software Development . . . 13
- 1.3 The DRAGON Project . . . 13
- 1.4 Supermarket Control System . . . 14
- 1.5 Structure of Thesis . . . 16

2 Object-Oriented Programming 17
- 2.1 Object-Orientation . . . 17
  - 2.1.1 Classes . . . 18
  - 2.1.2 Object-Oriented Design . . . 19
- 2.2 Object-Oriented Languages . . . 20
  - 2.2.1 "Pure" Languages . . . 21
  - 2.2.2 Polymorphism and Dynamic Binding . . . 22
  - 2.2.3 Static Typing . . . 23
  - 2.2.4 Inheritance . . . 24
- 2.3 Classification Schemes . . . 26

3 Introducing DRAGOON 29
- 3.1 Programming Philosophy . . . 29
  - 3.1.1 A Mixed Paradigm . . . 31
  - 3.1.2 Outline Design of the Supermarket Control System . . . 34
- 3.2 Template Packages . . . 34

3.2.1 Templates in the Supermarket Control System 36
3.3 Objects and Classes 37
3.3.1 Clientship 38
3.3.2 Class Bodies 40
3.4 Inheritance 41
3.4.1 Method Modification 44
3.4.2 Multiple Inheritance 46
3.5 DRAGOON Typing Scheme 48
3.5.1 Class and Data Types 50
3.5.2 Type Compatibility 51
3.5.3 Conformance 52
3.5.4 Subtypes 53
3.5.5 Terminology 54
3.5.6 Type Conversion 56

4 Software Reuse **59**
4.1 Component Engineering 59
4.1.1 Component Families 61
4.2 Component Hierarchies 62
4.2.1 Iterator Queue 64
4.2.2 Bounded Queue 66
4.3 Abstract Classes 68
4.3.1 Pre/Post Conditions 69
4.3.2 Completing Methods 70
4.4 Genericity 73
4.4.1 Constrained and Unconstrained Genericity 75
4.4.2 Inclusion versus Parametric Polymorphism 77
4.4.3 Declaration by Association 80
4.4.4 Parameterisation Contract 82
4.4.5 Inheriting Genericity 84
4.4.6 Type Compatibility of Generics 85
4.4.7 Method Overloading 86
4.5 Dynamic Binding 87
4.5.1 Extensibility 89
4.6 Programming by Difference 91

5 Concurrency **93**
5.1 Approaches to Concurrency 93
5.2 Concurrency in Object-Oriented Languages 97

5.2.1 Exclusion Synchronisation and Inheritance 100
5.2.2 Separating Synchronisation and Functionality 101
5.3 Behavioural Inheritance 103
5.3.1 History Functions and Deontic Operators 103
5.3.2 Behavioural and Behavioured Classes 105
5.3.3 The Exclusion Symbol 108
5.3.4 Behavioural Class Libraries 109
5.4 Active Objects 111
5.4.1 Threads 112
5.5 Agents 114
5.5.1 Condition Synchronisation 115
5.6 Forwarder Objects 118
5.6.1 Guarded Permissions 121
5.6.2 Guard Evaluation Semantics 123
5.7 Building Concurrent Systems 125
5.7.1 Reconfiguration 127

6 Distribution **129**
6.1 Loosely-Coupled Systems 129
6.1.1 Object-Orientation and Distributed Systems 131
6.2 Programming Distributed Systems 132
6.2.1 Separation of Concerns 133
6.2.2 Characteristics of Virtual Nodes 134
6.2.3 Suitability of Objects 135
6.2.4 Reference Free Communication 136
6.3 Virtual Node Objects 137
6.3.1 Virtual Node Construction Rules 141
6.4 Virtual Nodes in the Supermarket System 142
6.5 Executable Objects 145
6.5.1 "Heavyweight" and "Lightweight" Objects 148
6.5.2 Execution Support Classes 148
6.6 Physical Node Objects 151
6.7 Software and Hardware System Components 153
6.7.1 Physical Node/Execution support Pairs 156
6.7.2 Device Drivers 157
6.8 DRAGOON Execution Model 161
6.8.1 Centralised Reconfiguration 164
6.8.2 De-centralised Reconfiguration 168

7 Translation into Ada **169**
7.1 Possible Implementation Strategies . . . 169
7.1.1 "Abstract State Machine" Packages . . . 169
7.1.2 Tasks . . . 170
7.1.3 Abstract Data Types . . . 171
7.1.4 DRAGOON Translation Strategy . . . 175
7.2 Representation of Object State . . . 176
7.2.1 The CREATE Function . . . 179
7.2.2 The PART_OF Function . . . 180
7.2.3 Linear Inheritance . . . 181
7.2.4 Clientship . . . 184
7.3 Dynamic Binding . . . 185
7.3.1 Method Selection Shells . . . 186
7.3.2 Multiple Offspring . . . 189
7.3.3 Abstract Classes . . . 191
7.4 Multiple Inheritance . . . 192
7.5 Generic Classes . . . 196
7.5.1 Inheriting Genericity . . . 198
7.6 Concurrency . . . 200
7.6.1 Behavioured Classes . . . 202
7.6.2 Guarded Permissions . . . 205
7.7 Executable Objects . . . 207
7.8 Distribution . . . 209
7.8.1 Supporting Distributed Execution . . . 210

8 Conclusion **215**
8.1 Other aspects of DRAGON . . . 215
8.2 Contribution of DRAGOON . . . 217

References **219**

Chapter 1

Introduction

Developing software for embedded systems, in which the computers to be programmed are responsible for controlling larger, encapsulating systems, is one of the most difficult and challenging fields of software engineering. Not only is this type of software required to respond rapidly and reliably - in "real-time" - to events in the external environment, but it often has to execute under severe hardware and memory constraints. Moreover, embedded systems can be extremely large, and tend to require sizable teams of people to program them. In this application domain, therefore, the typical problems of software engineering are greatly complicated by considerations of concurrency, reliability, efficiency and software integration.

Early techniques for handling these problems tended to be based on a combination of ad hoc language modifications and operating system "fixes". The proliferation of language dialects that inevitably resulted soon fragmented the programming community, and reduced the prospects for sharing expertise and software between different projects. This, in turn, perpetuated the low productivity of programmers, and the poor quality of software, at a time when the hardware for embedded systems was becoming ever more powerful and cheap. As a result, manufacturers and users of embedded systems have been forced to place increasing emphasis on "*reuse*" at all stages in the software lifecycle, in order to combat the escalating cost of its development, and its generally poor quality.

At the same time the advances in microprocessor and networking technology also fuelled a trend towards "*distribution*", adding a whole new dimension to the software construction problems. Distributed systems have a number of potential advantages over their uni-processor counterparts - not only do they provide a cheap, flexible source of processing power, but offer also significant reliability improvements because of their resilience to hardware failure. Moreover, they introduce the possibility of "*dynamic reconfiguration*", in which parts of a system may be modified whilst it is running. Taking full advantage of these additional possibilities, however, creates

extra demands and difficulties for the software engineer.

The net result of these trends over the past two decades has been to greatly compound the traditional problems of developing software for embedded systems with the requirements of software reuse and distribution. One of the most pressing needs of the developers of large-scale industrial software systems, therefore, is for a uniform framework which not only addresses these traditional software engineering problems, but also encourages the reuse of existing software components in the development of new applications, and takes advantages of the prospects for dynamic reconfiguration and fault tolerance offered by distributed systems.

1.1 The Role of Ada

By far the most ambitious response to this so called "software crisis" in the embedded systems industry was the development of Ada for the United States Department of Defence [DoD, 83]. Its goal was to replace the vast number of languages and ad hoc techniques previously in use with a standard programming paradigm that could be used throughout the industry as a vehicle for promoting software reuse and programmer productivity. Out of necessity, therefore, Ada is a large, "broad-spectrum" language, providing constructs to embrace all the well established principles of software engineering as well as the special features needed for embedded systems (e.g. multi-tasking, exception handling etc.). In addition, Ada is one of the first languages containing special facilities for overcoming the traditional incompatibility of strong typing and reuse (generics). However, by far the most important feature of the language is the fact that it is a strictly protected standard - no supersets or subsets may take the name Ada.

Unfortunately, Ada was developed at a time when the full significance of distribution had not been fully appreciated. It is consequently poorly designed for the purpose of programming such systems, although the need for distribution is acknowledged in the Ada Language Reference Manual [DoD, 83]. Furthermore, the introduction of Ada has not turned out to be the panacea for reuse that was originally hoped. While, the language has certainly been the catalyst for significant advances, the expected explosion in the practice of reuse, and emergence of "software component factories" [McIlroy, 69] has failed to materialise to any significant degree. Generics, in particular, have proved more difficult to use in practice than was envisaged, with many different guidelines needing to be followed to ensure their effective use in modelling reusable components.

1.2 Object-Oriented Software Development

One of the most promising vehicles for reuse to have emerged in recent years is the rapidly expanding discipline of "object-oriented programming". Meyer [88] and Cox [86], in particular, have shown how the features of inheritance, dynamic binding and polymorphism offered by this paradigm provide an extremely powerful and elegant approach to reuse, which differs fundamentally from other mechanisms. Certain aspects of this approach can be helpful in the construction of Ada software, and there are now quite a number of "design methods" which exploit its basic structuring concepts to impose a discipline on the use of Ada. The language does not, however, directly support the inheritance, polymorphism and dynamic binding mechanisms needed to exploit the object-oriented approach fully.

Existing object-oriented programming languages have their drawbacks as well, however. No previous language has successfully extended this paradigm to handle concurrency and distribution without sacrificing much of its support for reuse. Object-oriented languages which handle concurrency or distribution, such as POOL [America, 87] or Emerald [Black et al, 87], do not support an inheritance mechanism permitting the "reuse" of implementations, while those designed to enhance the reuse advantages of the object-oriented approach, such as Eiffel [Meyer, 88], do not tackle concurrency or distribution.

1.3 The DRAGON Project

Despite its deficiencies in connection with software reuse and distribution, Ada is still of unique importance in the field of embedded systems. No other comparable language has a chance of matching its stability, portability or scale of industrial backing, and without a high degree of standardisation strategies for supporting reuse are of little more than academic interest.

Although languages have been designed, therefore, which are much more powerful than Ada for the separate purposes of reuse (e.g. Eiffel), or distribution (e.g. CONIC [Kramer and Magee, 85], Argus [Liskov, 82]), these do little to improve the *overall* prospects for software development in the embedded systems industry. Ada was introduced specifically to counteract the proliferation of specialised languages for particular applications, because this alone is a far stronger force *against* reuse than any features a particular language may provide to support it. What is urgently required is a language which combines the portability and stability advantages of Ada, the reuse mechanisms of object-oriented languages such as Eiffel, and the support for software distribution provided by specialised language such as CONIC

or Argus, within a unified framework.

To try to meet this need the European Esprit project DRAGON, has developed a language, called DRAGOON (Distributable, Reusable Ada Generated from an Object-Oriented Notation) which supports all the major features of the object-oriented paradigm, but extends this framework to handle concurrency, distribution and reconfiguration. DRAGOON can be translated into standard Ada for execution, and consequently can be regarded as a design language for Ada, but since tools have been developed to perform the transformation automatically, DRAGOON can also be regarded as a language in its own right.

So that experienced Ada programmers would feel comfortable with the language, all features of DRAGOON alien to Ada have been designed to fit in with the Ada spirit and philosophy. In one sense, therefore, DRAGOON perhaps indicates how Ada might have turned out had the importance of distribution and advantages of object-oriented programming been apparent when it was designed, and in the long term may provide ideas for the future development of Ada.

The purpose of this thesis is to introduce the main features of the DRAGOON language, and the rationale behind them. This will not be achieved in the style of an exhaustive reference manual, however, since such a document already exists [Di Maio et al, 89], but by means of a case study illustrating how the software for a simplified real-time system might be developed in the language.

1.4 Supermarket Control System

The system considered is derived from the "Supermarket Control System" described in the final report of the DIADEM project [Atkinson et al, 88], which is a simplified version of the type of system used to control the flow of products through a large supermarket or department store. Its main function is to maintain a *database* describing all the products sold in the supermarket so that the *checkout points* distributed throughout the store can establish the prices of the items selected by the customers. These items are identified at the checkout points by means of a laser-readable bar code.

Storing all product information in a single database, rather than distributing it around the supermarket, simplifies the problem of maintaining consistency, and enables it to be updated from a single *central control point.* This provides the interface to the human managers of the system. As well as giving them access to the information stored in the database, such as the number of items of a given product type currently in stock, it allows them to modify, add or remove the *product descriptors* describing these attributes.

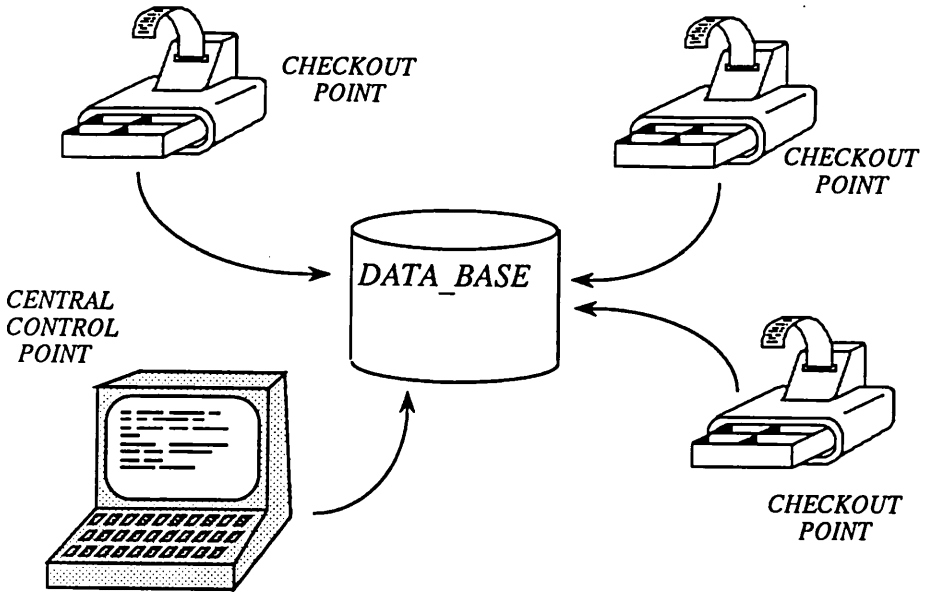


Figure 1.1: Components of the Supermarket Control System.

Each type of product sold in the supermarket has four pieces of associated information :-

- the unique "bar-code" number,
- the price,
- the number of items of the product type currently in stock,
- a textual label storing such information as the product name, manufacturer, country of origin etc.

In addition to these attributes, perishable products, such as foods, have an associated "sell-by-date" after which time they are no longer fit for sale, while electrical products have an associated fuse rating, describing the size of the fuse needed for their adequate protection.

A real system of this kind would naturally handle many more functions of the supermarket. The DIADEM version, for example, also dealt with aspects of security. However, the simplified view of the system described above is of sufficient complexity to illustrate all important aspects of the DRAGOON language. Any additional requirements will be added where necessary in later chapters.

In the form described above, and illustrated in Figure 1.1 the system could be implemented using a single, centralised computer to house the database, and control the other components (i.e. the control and checkout points) as remote devices, or alternatively as a distributed system with a dedicated machine for each component.

1.5 Structure of Thesis

The thesis assumes a basic working knowledge of Ada, but in the following chapter provides a general introduction to object-oriented programming for those unfamiliar with this style of software construction. This chapter may be skipped by the expert reader. Chapter 3 introduces the DRAGOON language, and describes how the principal mechanisms of object-oriented programming identified in chapter 2 are realised in an "Ada oriented" fashion.

The next three chapters concentrate on different aspects of DRAGOON from the perspective of the three major themes of this thesis - software reuse, concurrency and distribution. Chapter 4 describes how DRAGOON harnesses and enhances the powerful reuse features of the object-oriented paradigm, and how the apparently unrelated concept of genericity is exploited. Chapter 5 then goes on to discuss the concurrency features of DRAGOON, describing how the conventional inheritance model is extended to handle synchronisation requirements. DRAGOON's strategy for supporting the construction and dynamic reconfiguration of distributed systems within the object-oriented framework is then analysed in chapter 6.

Once the overview of the language is complete, chapter 7 describes how DRAGOON's features are implemented by automatic translation into Ada. Chapter 8 then concludes by providing a brief overview of other supporting aspects of the DRAGON project, and by describing the contribution made by DRAGOON to software engineering technology.

Chapter 2

Object-Oriented Programming

The term "object-oriented" is now so ubiquitous that no one is quite sure what it means any more, particularly in the field of software development where many widely differing languages, environments and design methods lay claim to this description. Although various attempts have been made to identify and classify the essential properties of the object-oriented philosophy [Wegner, 87], [Blair et al., 89], the provision of a broad, formal foundation for this model of computation is still some way from realisation. It is impossible, therefore, to provide a precise and complete definition of object-oriented programming, but this chapter attempts to allay potential confusion by explaining the different contexts in which the term is used, and by identifying the concepts generally regarded as characterising object-oriented systems.

2.1 Object-Orientation

Although new "declarative" styles of programming have emerged in recent years to challenge the traditional "imperative" view of software execution, the fundamental structuring concepts of software engineering are still pertinent and valuable tools for guiding the design process, and enabling humans to cope with the complexity of large applications. Chief amongst these are the principles of *modularity* - splitting a design up into smaller, more manageable components that can be tackled separately; *abstraction* - separating the concerns for what a particular component does from how it does it; and *information hiding* - concealing detail irrelevant at a particular level of abstraction. Because of their importance, most modern languages directly support at least some of these principles, but often by means of distinct and poorly related constructs. Ada, for example, provides packages as modules for grouping logically related declarations, but relies on the additional concept of private data types to enforce abstraction and information hiding. Much of the current success of the

"object-oriented" view of software construction can be attributed to the fact that it is based on a simple, natural concept through which all these principles are unified - the *object.*

Whereas traditional software design strategies focus on the procedural steps a program must perform to achieve the desired goal (i.e. on algorithms), and so tend to draw a distinction between passive data and active operations, the object-oriented approach views a program as a "model" of the system composed of a set (or federation) of interacting "objects". These objects integrate data structures with the operations which act upon them, and so give the data a much more significant role in determining the software architecture - an emphasis neatly captured in Meyer's [88] "motto" for the object-oriented approach :-

"Ask not first what the system does, ask what it does it to".

An object has a set of operations and a state which remembers the effect of those operations [Wegner, 87]. Moreover, an object permits its state to be manipulated only by the operations exported in a well-defined interface, thereby shielding details of its internal implementation[1], and ensuring that it interacts with others at the desired level of abstraction. In the object-oriented approach, therefore, the principles of data abstraction and information hiding are united with the natural modularity of objects into a single coherent framework.

The object-oriented approach, however, is more than just an elegant technique for encouraging modularity and abstraction in program design. It also tends to decentralise knowledge and data, so confining implementation details to those parts of the system which really need access to them. Conventional "algorithm oriented" methods, on the other hand, tend to produce hierarchical "master/slave" software structures assigning different levels of importance to different components. As a result software components higher up in the hierarchy require knowledge of implementation details of those they depend on. By confining these details to the operations which truly need to be aware of them the object-oriented approach limits the impact of changes, and makes the resulting software much more flexible and extendible. This will be discussed in more detail in chapter 4.

2.1.1 Classes

Since the object-oriented approach to program decomposition allows systems to be modelled in terms that more closely resemble the physical world, it is perhaps not surprising that it first arose in the field of simulation. Simula [Birtwistle et al., 83], a language designed primarily for discrete event simulation, is generally

[1] But see chapter 6.

credited with being the father of the object-oriented paradigm, introducing many of the key concepts. Perhaps the most fundamental of these is the idea of a *class* acting as a template, or blue-print, for the generation of numerous structurally and behaviourally identical objects, or *instances.* A class in Simula therefore can be thought of as an "object type".

The notion of classes is closely related to the concept of abstract data types as supported in most modern imperative languages and formalised using algebraic specification techniques. In fact, object-oriented design is often taken to mean simply the design of software using abstract data types [Meyer, 88] - a class being viewed as an implementation of an abstract data type. However, this interpretation of classes is valid for only a subset of the program abstractions which can potentially be modelled as objects. Booch [87] points out that, unlike abstract data types, certain classes of objects are characterised not merely by the the services they provide, but also by services they require, at the same level of abstraction, from other objects. This is particularly so in concurrent systems, where objects have the additional features of independent, concurrent "threads of control" and synchronisation constraints on the execution of their operations. In fact, one of the goals of this thesis is to investigate the adaptation and application of the notion of objects to other programming domains such as distribution and reconfiguration.

The interpretation of classes as abstract data type implementations is therefore valid only for sequential software systems, but is nevertheless of fundamental importance because it provides one of the few routes to a formal model of "object-orientedness", via the theory of algebraic specifications associated with abstract data types.

2.1.2 Object-Oriented Design

The idea of basing the architecture of a program on the "objects" manipulated in the application domain is a fundamentally different approach to software construction which transcends any particular programming paradigm or language. Consequently, the last few years have seen a surge in the popularity of "object-oriented" design methods for use in conjunction with all kinds of languages [Ormsby, 90]. Many of the most recent, however, have been developed specifically for Ada as the target language, since it supports most of the software engineering principles mentioned above, particularly modularity, which are necessary for taking full advantage of this approach.

Although the use of classes and objects, or their equivalent, as the basis for system architecture is found in many earlier design strategies, such as MASCOT [2],

[2]which can now be targeted to Ada.

one of the first people to advocate the use of "object-oriented" techniques, as such, for the design of Ada programs was Booch [83]. His work has been extended by a number of other object-oriented design schemes such that of EVB [Berard, 85] and [HOOD, 87]. As pointed out by Booch, however, object-oriented design methods deal with only part of the lifecycle of a typical piece of software, specifically, the step of transferring the abstract requirements into code. Consequently they usually need to be used in conjunction with requirements elicitation and analysis techniques such as JSD [Jackson, 75], CORE [Mullery, 76] or SCS [Potts et al., 86].

2.2 Object-Oriented Languages

Although emphasis can be placed on structuring in terms of objects without any special linguistic support, object-oriented programming in its full generality requires the use of specialised languages. Following the lead of Simula, there is now a large number of languages designed specifically to embrace this style of programming, and incorporating special constructs for modelling objects. In most cases these object-oriented facilities serve as high-level devices for imposing structure on software written according to some other programming paradigm. There are, consequently, examples of "object-oriented" enhancements to all the major schools of programming. LOOPS [Brobow and Stefik, 82] and Flavors [Moon, 86], for example, are languages enabling code written in the classic functional language LISP to be arranged and composed in terms of higher-level "objects". C++ [Stroustrup, 86] and Objective C [Cox, 86] are examples of languages performing similar roles with the imperative language C, and languages like ESP [Chikayana, 84] and POLKA [Davison, 89] superimpose the concept of objects on the logic programming languages Prolog and Parlog [Gregory, 87] respectively.

At first sight, the notion of state-encapsulating objects might seem to be in direct conflict with the fundamental tenet of declarative languages, which is the avoidance of destructive assignments to memory locations. However, the notion of an object state can be attained without destructive assignment by the recursive invocation of predicates or functions. Changing the parameters of the recursive call after each operation mimics the effect of a state change on the processing of the next operation.

The common feature of these so called "hybrid" languages is that they allow objects to be treated as "first class citizens". Thus, although objects are structuring units, encapsulating state and operations, they are also treated as basic data items which can be instantiated dynamically and passed as parameters to operations. In other words, objects serve both as the *operands* acted upon by operations, and as the *operators* performing operations on others. This duality at first is one of the

most confusing aspects of object oriented programming to newcomers, but also the basis of much of its power.

2.2.1 "Pure" Languages

Taken to its logical conclusion, however, the object-oriented approach is more than just a device for structuring code fragments written according to some conventional programming paradigm, but becomes a complete programming philosophy in its own right. Provided that a rich enough set of pre-defined primitive "objects" is available no other structuring concepts are needed to provide a fully general language capable of tackling most problems. Programmers are then truly only concerned with describing objects and there mode of interaction.

The first language to adopt this philosophy wholeheartedly was Smalltalk-80 [Goldberg and Robson, 83], which is consequently regarded as the archetypical object-oriented language, and is the origin of much general "object-oriented" terminology. More recent examples of "pure" languages adopting a uniform object model include BETA [Kristensen et al., 85], Emerald [Black et al. 86] and POOL [America, 87]. Smalltalk refers to the operations exported by objects as *methods* and introduces the idea of objects interacting by exchanging *messages.* This should not be confused with the notion of messages in concurrent or distributed systems, which reflects a particular implementation strategy for inter-process communication (chapter 5), but is used in this context to stress the independence of the request for the invocation of a method - the *invocation request* - from its actual execution.

Smalltalk, however, is also responsible for much of the confusion surrounding object-oriented programming, because it has a number of other unique features which are not really part of the object-oriented paradigm. The programming language is, in fact, only one part of a fully integrated programming and execution environment providing a high quality user interface, and for many years these equally innovative aspects of Smalltalk obscured the power of its object-oriented programming features.

The fact that Smalltalk is a combined development and execution environment allows it to treat classes and objects in a way that is impossible in "compiled" object-oriented languages such as Simula, Eiffel or C++. In Smalltalk, classes are not regarded as merely static templates defining the structure of the various object that can exits at run-time, but also as objects which themselves have a run-time existence, and can respond to messages just like any other. Each Smalltalk class is an instance of a *metaclass*, which in turn is an instances of another metaclass, and so on up to the class *object* which represents the root of the entire Smalltalk object hierarchy. For consistency this object is viewed as an instance of itself.

Adopting a completely uniform view in this way, in which all classes are also objects, has a number of advantages as far as the consistency and power of the model is concerned. Since every object in the system is an instance of a class, which is itself a run-time object, the class may store sharable information common to all the instances. *Class variables* act as global data accessible to all instances of the class, while *class methods* enable operations such as instance creation and destruction to be defined as normal methods of the class. Creation of an object is thus modelled as sending a "create" message to the object's class. In languages such as Simula, Eiffel and POOL, in which classes do not have a run-time existence, the creation and deletion of objects has be handled by special predefined operations, or "allocators", of the language, usually called "create" or "new". Consequently, the definition of other class methods, or reimplementation of the "create" operation, presents certain conceptual difficulties in such languages. In POOL, the equivalent of Smalltalk class methods are called *routines* to distinguish them from the normal methods which form the callable interface of instances.

While this uniform object-oriented model has a certain elegance, it is inappropriate for typical software development purposes in which the run-time environment must be regarded as distinct from the development environment. Most compiled object-oriented languages, therefore, retain the traditional Simula view of classes as purely static entities. Objective C [Cox, 86] has an interesting compromise which lies somewhere in between the purely static and purely dynamic view. Classes themselves are not regarded as existing at run-time, but give rise to special *factory objects* which act as their run-time surrogates capable of responding to the equivalent of class methods.

2.2.2 Polymorphism and Dynamic Binding

Whether or not the creation of objects is performed by a class method, or a predefined language operation, most object-oriented languages allow objects to be generated dynamically. Since the number of objects is therefore unpredictable at compile time they have to be identified indirectly by references rather than static names. These references are stored in program elements usually termed *instance variables.* Since several of them may refer to the same object at a given moment in time, they are also sometimes known as *aliases* for the object concerned.

Instance variables may be made to reference instances of different classes during the life of a program, and so are polymorphic entities - they are able to denote several different forms of object. In itself this facility is no more powerful than conventional techniques for achieving polymorphism, such as variant records in Pascal and Ada, but combined with the object-oriented view of method invocation it gains much more

significance. Since different classes may have different implementations of a given method (i.e. the method name can be overloaded by different classes), the particular version of the method which will be executed in response to a given request depends dynamically on the class of the object. In other words, the name of the operation appearing in the invocation request is dynamically bound to the particular version exported by the object referenced by the instance variable.

Dynamic binding of this kind is one of the most important features of object-oriented programming since it explicitly separates the act of requesting a service from that of deciding which method will be executed to satisfy it. The former is conceptually the responsibility of the client and the latter of the server, but most conventional programming techniques require the client to perform both.

2.2.3 Static Typing

Certain languages such as Smalltalk support polymorphism and dynamic binding of this form without constraint, and allow any object to be assigned to any instance variable. There is no guarantee, therefore, that the method named in an invocation request will be possessed by the receiving object. If it does not possess a suitable method, the infamous "message not understood" message is displayed and no operation is invoked.

Since it avoids the possibility of data corruption, so called "dynamic typing" of this kind is preferable to no typing at all, and is acceptable in the prototyping applications for which Smalltalk is primarily intended. It is not acceptable, however, for applications in which reliability is a major concern and in which it is desirable to have some assurance before run-time that objects will be able to service invocation requests directed to them. To achieve this many object-oriented languages employ static typing mechanisms to constrain the assignments allowed on instance variables.

The idea of the typing schemes in most modern languages is to assign a "type" to all the values in a program, defining the set of operations and contexts in which they may be used. Any attempt to use a value in a way that contradicts its declared type can thus be detected by the compilation system, and the program rejected as erroneous. The concept is usually extended to include the notion of subtypes which identify values that possess at least all the properties of the parent type, and which therefore may be used in contexts where a value of the parent type is expected.

The corresponding idea is applied to classes in the object-oriented programming paradigm by the general notion of *conformance*. One class is said to conform to another if it can be used in all contexts where the other is expected, that is - if it can understand and respond to all the invocation requests (i.e. messages) handled by the other. This essentially means that in the interface of a class - its protocol

- subsumes that of any class it conforms to. Conformance, therefore, depends only on the protocols of classes, defined by their interfaces, and is not in any way effected by details of their implementation. In many languages, such as Emerald and Trellis/Owl [Shcaffert et al., 86], conformance is used as the sole basis for defining *type compatibility* - the rules determining when an instance of one class may be assigned to an instance variable of another - but in other languages, such as Eiffel, classes must also have implementations related by an *inheritance* mechanism to be type-compatible.

2.2.4 Inheritance

Inheritance is another idea first introduced in Simula (but under the different name of concatenation) and is generally regarded as one of the most important features of object-oriented programming. Essentially it is a mechanism for sharing knowledge in systems by enabling new classes to share (i.e. reuse) the implementations of those already existing. In other words, it allows the programmer to define new classes by explicitly stating how they differ from existing ones, rather than by specifying all their properties from scratch. This is one of the main ways in which the object-oriented paradigm promotes reuse, and will be discussed further in the following chapters.

Using inheritance, a new class, the *heir*, may be defined as a specialisation of another class, the *parent*[3], by inheriting its methods and instance variables, and adding to them. The heir class therefore usually conforms to its parent, since it normally inherits all the methods in its interface. In many languages this need not necessarily be the case, however, since heirs can remove inherited methods from their interface, and redefine the parameter types.

There are several different variations on the inheritance theme. Inheritance may be *single*, in which case a class may only have one parent, or may be *multiple*, in which case it may have many. Alternatively, depending on the view of classes in the system, inheritance may be *static* or *dynamic*. In compiled languages such as Simula, Eiffel and C++, inheritance is static since all information sharing is fixed at compile time. Systems such as Smalltalk, however, in which all objects in the system are accessible at run-time, provide a dynamic inheritance mechanism. The method which will be executed in response to an invocation request is located dynamically by the Smalltalk system. It first searches for a suitable method in the class of the object which received the request, and if unsuccessful looks at the superclass, and so on up the inheritance hierarchy. This allows the user to modify inherited methods

[3]Many languages, including Smalltalk, use the term *subclass* and *superclass* synonymously with "heir" and "parent" (See Section 3.5)

while instances of inheriting classes are executing. The changes will become known to instances of the descendent classes on the next invocation of the method.

Deferred Implementation

Straightforward inheritance enables the methods of new classes to reuse the implementation of methods of existing classes. However, it is also possible in most object-oriented languages for the reverse dependency to exist, in which the implementation of "old" methods may be defined by "new" classes. In other words, the designer of a class may entrust the implementation of its methods to classes yet to be defined. This is achieved in compiled languages such as Simula and Eiffel by the declaration of special methods whose implementation the compiler recognises as being *deferred* (or virtual in Simula).

In dynamic languages such as Smalltalk, on the other hand, this effect is achieved by means of the special pseudo-variable *self*. This does not, as its name might imply, necessarily refer to an instance of the class in which the code appears, but rather to the object first receiving the invocation request, which may be an instance of a descendent class. Thus, if an object receives a request for the invocation of an inherited method, during the execution of the method "self" will refer to this object, which first received the request, not an instance of the class in which the method is originally defined. Therefore, the search for methods invoked via "self" starts from the class of the receiving object.

The possibility of defining *abstract* classes, some of whose methods are yet to be implemented, enables systems designers to capture parts of their design in source code before the whole picture is complete. Object-oriented languages can therefore be used earlier in the development process than many other more traditional languages, and so can in some ways also be regarded as design notations [Meyer, 88].

Delegation and Prototypical Objects

In Smalltalk the search for method code is performed "invisibly" in the background by the environment. However, since classes are regarded as run-time objects, using the idea of the "self" variable it is possible to view this searching activity as part of the normal interaction of objects. This gives rise to the delegation model [Lieberman, 86], [Borning, 86], and essentially means that when an object receives an invocation request for a method it does not itself directly implement, it *delegates* the servicing of this request to another object - its parent. The locating of methods to service messages is therefore explicit, rather than implicit. Delegation involves more than

just the forwarding of a message, however, but also the binding of any references to "self" in the method to the delegating object.

When method searching is modelled using the concept of delegation the notion of distinct classes and instances loses much of its value. It is more natural to combine the functionality originally separated into a class and instance into a single object, and model the creation of many identical clones as copying the original, rather than instantiation of a template. The delegation model is therefore usually combined with the notion of "prototypical" objects [Lieberman, 86] which represent both normal active components of the system (c.f. instances) and blue-prints for the creation of others (c.f. classes). When an objects is generated as an adaptation of a prototypical object, it delegates the servicing of the common ("inherited") methods to the prototype (its parent).

It has been claimed that the delegation/prototypical object model offers a more natural and powerful technique for representing relationships between objects than the traditional class/inheritance model [Lieberman, 86]. However, it is now widely recognised that there is little, if any, difference in expressive power of the two approaches. Delegation is in fact a widely used technique for implementing class-based inheritance.

2.3 Classification Schemes

Although "object-orientation" became a recognised software structuring technique in its own right less than a decade ago, the ideas outlined above are now finding application in almost all spheres of computing. To help categorise the different ways in which these ideas are employed, it is usual to distinguish the term object-oriented *design*, which essentially implies an approach to program decomposition based on the notion of classes and objects, from object-oriented *programming*, which implies the use of a specialised language. In principle, object-oriented design does not rely on any specific linguistic facilities, having even been deployed for the development of FORTRAN programs, but it is mainly used in connection with modular languages such as Ada, and Modula-2.

Whatever linguistic facilities are available, however, both object-oriented design and programming are essentially concerned with employing the structuring techniques for the *development* of application software. The "object-orientedness" of the eventual executing software is not of particular interest. Object-oriented *environments*, on the other hand, are not so much concerned with the language or methodology used to construct software components, but with employing the object-oriented structuring techniques for organising running systems. Many oper-

ating systems and data bases [Banerjee et al., 87] for example, exploit the notion of objects for handling the execution of software and storing data. As with object-oriented design methods, the construction of object-oriented environments can be undertaken without a specialised language, but is greatly eased by their availability.

The range of different language features captured under the umbrella of "object-oriented programming" is enormous. Perhaps the simplest and most well-known scheme for classifying these various languages is that of Wegner [87], which is based on the language mechanisms discussed above. To earn the title "object-oriented" in Wegner's classification a language must provide support for inheritance, as well as features supporting the definition of classes and objects. Those languages supporting the class/instance distinction, but with no form of inheritance mechanism, are termed "class-based" languages, while those which lack also the concept of classes, and merely provide support for encapsulated objects, are termed "object-based". Wegner goes on to define further categories of languages depending on their support for strong typing, concurrency and distribution. These aspect will be considered in later chapters.

Blair et al [89], on the other hand, have proposed an alternative approach which concentrates more on the essential *properties* of object oriented systems than on specific mechanisms. The properties of object oriented systems are analysed according to three criteria - encapsulation, abstraction and polymorphism.

Neither of these classification schemes, however, successfully captures one of the major differences between the various object-oriented languages, which is their view of classes at run-time. Smalltalk, for example, views classes as *dynamic* (i.e. run-time) objects, while most of the "compiled" languages view them as *static* (i.e. compile time) templates. A comprehensive classification scheme needs to embrace all three aspects of "object-orientation" outlined above (i.e. "design", "environments" and "languages"), since Smalltalk's model of objects depends on the fact that it is both a language and an environment.

Chapter 3

Introducing DRAGOON

This chapter introduces the language DRAGOON, and describes how it supports many of the object-oriented programming features outlined in the previous chapter. Naturally, the language which has had the greatest influence on its features is Ada - one of the basic design goals of DRAGOON being to remain as faithful to Ada style and syntactic conventions as possible. Another important influence on the language, particularly its "object-oriented" features, has been Eiffel, developed by Bertrand Meyer [88], also with the aim of exploiting the paradigm's power for reuse. Although heavily disguised in Ada-like syntax, DRAGOON adopts Eiffel's basic model of classes and objects, and its strategy for expressing method calls.

As another important design criterion of DRAGOON was that it should be translatable into straightforward, intelligible Ada, it also has much in common with other object-oriented design techniques targeted to Ada. Some of these, therefore, particularly HOOD [87], and the work of Booch [87], have also influenced DRAGOON's features.

3.1 Programming Philosophy

As described in the previous chapter, the basic philosophy of "object-orientation" is to base the architecture of software systems on the tangible objects in the problem domain. The most fundamental requirement of languages designed to support this approach, therefore, is for some means of modelling the objects in a system. In common with most other object-oriented languages DRAGOON adopts the "class" as the mechanism for achieving this. Programming in class-based languages, such as DRAGOON, is largely concerned with the definition of new classes in terms of instances of those defined previously, until ultimately the entire system is modelled by one or more classes. An extreme example of this object-oriented approach is embodied by Smalltalk, in which *every* entity in the problem domain must be modelled

as an "object".

Many of the concepts which need to be represented within computer programs, however, are not always naturally modelled as objects [MacLennan, 82]. Immutable abstractions without substance or location, such as colours and numbers, are much more intuitively thought of as "values" than objects. The quality of "greenness", for example, which would probably feature heavily in a traffic-light simulation program, is not intuitively an object, since it can neither be copied nor counted. Greenness is a *quality* (or attribute) of objects, not an object in itself. The same is true of numbers. Although the number "two", for example, can be represented physically by the strings "two" or "2", the abstraction itself is a "pure" mathematical concept without substance or location. Consequently, it can (and frequently does) exist solely in people minds whenever they associate a pair of objects together.

For abstractions of this nature, it is important to distinguish between the pure concepts themselves and their various possible *representations.* A representation of a number in a computer, whether at the source or machine level, is certainly an object and can be shared where necessary for efficiency reasons, but this does not alter the fact that the abstraction itself is not intuitively an object. In spite of this, some languages such as Smalltalk, in their quest for a completely uniform object model, do attempt to treat value abstractions (not merely their representation) as objects. This is achieved by regarding such values (e.g. integers, characters and boolean values) as special "immutable" objects which can never change their internal state. They consequently lack many of the fundamental properties usually associated with objects - they cannot be copied like normal objects, and instances can not be dynamically created and destroyed. All the instances of the classes **Character**, **SmallIntegers** and **Symbol**, for example, are generated at system initialisation time, and are assumed to remain in existence indefinitely.

The restrictions placed on immutable objects in Smalltalk essentially constrain them to behaving as values. At best, therefore, attempting to describe such abstractions within a uniform object-based metaphor provides no tangible advantages over the value-oriented approach, and at worst greatly confuses and complicates their manipulation. It certainly places an uncomfortable asymmetry on simple arithmetic operations such as addition. Although conforming to the usual syntactic style, an arithmetic operation in Smalltalk is conceptually performed by sending a message containing the second operand to the first, so that the first can add itself to the second.

Because the traditional "value-oriented" view of arithmetic operations is much closer to our arithmetic intuitions, few object-oriented languages have followed the pure Smalltalk approach, but prefer instead to handle numerical values and charac-

ters as traditional data types.

3.1.1 A Mixed Paradigm

Instead of dogmatically pursuing a pure "object-oriented" or a pure "value-oriented" approach, software development becomes much more straightforward and manageable if the logical distinction between objects and values is explicitly recognised [MacLennan, 82]. Each approach provides important advantages in certain contexts, but is inappropriate in others. Consequently, one of the basic principles of DRAGOON is to provide a "mixed" programming paradigm which allows a natural and complementary treatment of both "values" and "objects", and which is thus more powerful than either of these paradigms individually. DRAGOON, therefore, *supports* object-oriented programming but does not *enforce* it.

In conventional strongly typed languages such as Ada, abstractions that need to be treated as "values" are often represented by simple enumeration types. For example :-

```
type COLOUR is (RED, AMBER, GREEN);                    /3.1/
type PRODUCT_TYPE is (PERISHABLE, ELECTRICAL, BASIC);
```

Not all "value abstractions" however, have such simple representations in computers. Consider, for example, how *dates* are handled. Although dates are essentially just natural numbers enumerating successive days, because humans like to organise their lives according to the motion of the Earth, the representation of a date in a computer is usually described by three components - the day, month and year. Nevertheless conceptually, like numbers, dates are value abstractions, because a given date is a fixed and uncopyable notion[1]. To provide full flexibility in the representation of complex "value abstractions" such as dates therefore, DRAGOON permits the use of all Ada's typing facilities, rather than just primitive types.

Few other languages recognise and attempt to support both the "value oriented" and "object-oriented" paradigms in their full generality. Most object-oriented languages which support any form of "value-oriented" programming do so only for the primitive data types (e.g. *INTEGER*, *CHARACTERS* and *REAL*), while conventional imperative languages such as Ada tend to focus exclusively on the value-oriented view. As a consequence the term "object" is often used in different ways in different languages. In Ada, for example, any data structure brought into existence by the elaboration of a declaration, or the execution of an "allocator" operation, is called an object. Moreover, no explicit distinction is usually made between static objects

[1]Is it meaningful to have two copies of the *abstraction* "5th December 2010", for example?

generated in this way, and the variables or "locations" in which they are stored. The decision as to whether the identifier denotes the object, or the location in which it is stored, is made from the context in which it appears. When the identifier features as the source of an assignment it is taken as referring to the object, while when it features as the target, it is the location.

In class-based object-oriented languages, on the other hand, "object" has the special meaning of an instance of a class. Moreover, like dynamically allocated data structures in languages such as Ada and Pascal, the object is viewed as distinct from the variable, or location, which stores its reference (or internal name in Ada). To avoid the obvious potential for confusion in DRAGOON, which has all these kinds of data structure, it is the "object-oriented" meaning which is adopted and other more specific terminology used in place of the Ada meaning (e.g. array, integer, variable etc.). In DRAGOON, therefore, *all* objects are instances of classes.

Given a completely "mixed" paradigm of this form, the programmer must not only identify the meaningful abstractions in the problem domain, but must also judge which of them are best regarded as objects and which as values. Selecting the objects in a system is often a non trivial task, and is one of the weakest points of most object-oriented design methods. In the same way that one sometimes has to resort to using the nouns in an (informal) specification to identify the objects, and the verbs to identify the operations, the adjectives will often be a good indication of the "qualities" or values".

Not all the abstractions appearing in a problem will be so obviously recognisable as objects or values, however. Certain abstractions may have acceptable *representations* in either the value-oriented or object-oriented style. Dates are a good example. Although as a pure abstraction a data is a value, it may be represented in a program as an object which stores the three components (e.g. day, month and year) as more primitive values. Alternatively, in Ada, dates may be modelled as values by the definition of a package exporting a "date" type and a set of subprograms that operate upon it. Because DRAGOON supports both ways of modelling such abstractions, it is important to distinguish carefully between the two, and assign special meanings to the usual terminology.

A concept such as a date, with a set of possible values and associated operations, is commonly termed an *abstract data type* and in its pure form is independent of any representation or implementation strategy. In the majority of object-oriented languages the *class* is the only mechanism available for modelling such an abstraction, and consequently classes are often described as "implementations of abstract data types" Meyer [88], America [89]. Since DRAGOON also allows such an abstraction to be modelled in the traditional value-oriented style using a package, to distinguish

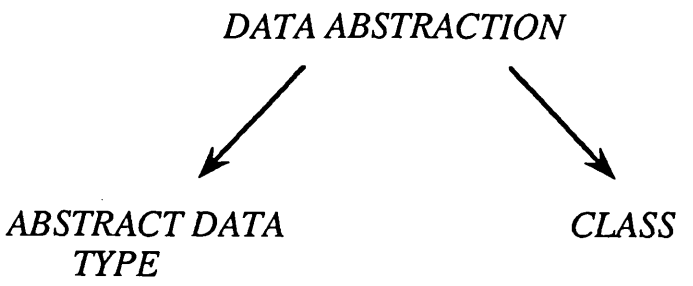


Figure 3.1: Alternative Implementations of a "Data Abstraction"

between these two possible implementation approaches a class is *not* regarded as an implementation of an abstract data type. This term is reserved for the traditional Ada style representation in which a package exports a (data) type and a set of associated operations which manipulate values of that type. If it is necessary to distinguish the pure concept from one of its two possible representations (i.e. classes or abstract data types) the "pure" abstraction will be referred to as a *data abstraction*, as illustrated in Figure 3.1.

The question as to whether to model a particular data abstraction as a class or an abstract data type is largely one of granularity. Most languages, including object-oriented languages such as Eiffel, objective C, and C++, have a set of "primitive" or "simple" data types which are used so frequently that they have been built into the language. Even Ada, whose designers had a conscious policy of separating language specific features from operations and types defined using them[2], permits these primitive values to be used in any program unit without the use of explicit "*with*" clauses.

When the data abstraction being represented in DRAGOON can be regarded essentially as an addition to the basic set of "primitive" values likely to be used frequently throughout the program, then it is better to define it as an abstract data type. If, on the other hand, it is a more substantial component of the system, which is not likely to be used as a primitive type in other parts, then it is better represented as a class. Classes, therefore, are more commonly used for larger components in the system, unlikely to recur frequently in other components, whereas abstract data types are used for smaller enties, such as dates, which will tend to be used for defining the structure of classes. In other words, classes tend to be used as a facility for programming "in the large", while abstract data types are used to provide additional types used for programming "in the small".

[2]In Ada, for example, the I/O routines are not regarded as an integral part of the language, but as a set of predefined subprograms.

3.1.2 Outline Design of the Supermarket Control System

To use DRAGOON effectively it is important that the programmer is given some help in tackling these difficult design issues, which arise at the earliest, most crucial, stages of software development. The accompanying design method developed in the DRAGON project, DEMON, therefore provides guidelines in the respect. It is not the purpose of this current chapter, however, nor of the thesis in general, to describe DEMON and other possible design strategies in detail (full details of DEMON can be found in [Bott, 89]). Rather, as stated in chapter 1, the goal is to illustrate the main features of DRAGOON by demonstrating their use in the development of a small case study - the Supermarket Control System - described in the previous chapter. All the complex issues likely to be faced at the early stages of designing such a system, therefore, will be neglected, and the following overall choice of objects and values in the system will adopted as the starting point for development in DRAGON.

The description of the system given above strongly suggests that the data-base, the checkout points and the central control point are the principle objects in the system. The other contenders for representation as objects are the product descriptors which hold the information about products in the system. Although they do not correspond directly to physical objects, there generally being many items (i.e. objects) of a given product type for each descriptor, they are not qualities or "pure" abstractions such as "greenness" or "numbers", and can conceptually be shared and copied.

All of the information stored by the descriptors could be represented quite reasonably as values. However, since the purpose of the "fuse rating" associated with electrical items is to indicate which type of fuse it requires, and fuses are also presumably products sold in the supermarket, instead of recording the fuse rating as a simple numerical value, the information will be stored indirectly as a reference to a descriptor of the particular "fuse" product actually required. As well as making it easier to provide the customer with information about the product, this strategy also has the commercial advantage to the supermarket that the fuse is more likely to be purchased along with the electrical item needing it.

The data-flow between the various objects identified above is illustrated in Figure 3.2.

3.2 Template Packages

If the product attributes are to be represented as values of a conventional Ada data type, the corresponding type definitions must be contained in sharable modules

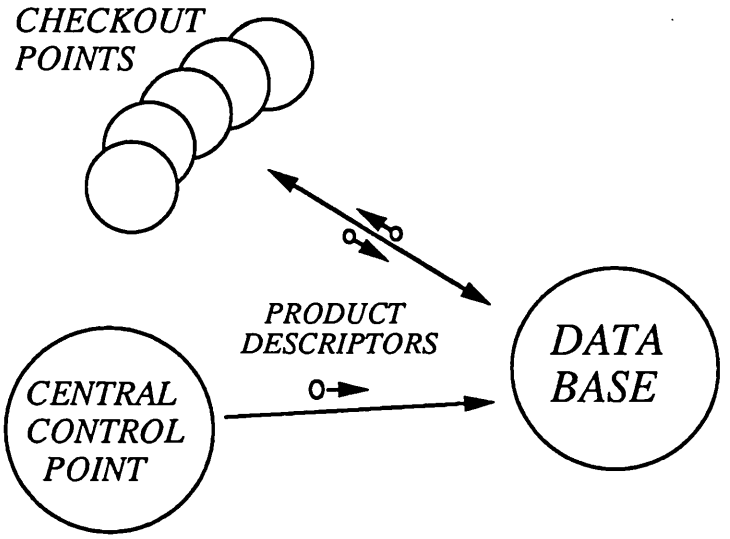


Figure 3.2: Data flow in Supermarket Control System

which can be made visible to any classes that need to see them. The obvious candidate for such a module is the Ada package, since one of its main purposes in Ada is to provide a convenient repository for grouping together logically related declarations. The package, therefore, has been adopted as one of the main structural elements of DRAGOON. Whereas normal Ada packages may declare state variables, however, and thus model abstract state machines, this is not desirable in DRAGOON because state is to be encapsulated by "objects" in the way described below. It is important that packages used in DRAGOON have no associated state whatsoever.

Packages with precisely this property were found to be useful in the DIADEM project [Atkinson et al, 88] for housing declarations shared by potentially distributed units. Such packages were termed "*template units*" since they mainly contain declarations acting as "templates" for the subsequent generation of variables. In fact, in DIADEM, any Ada library unit not possessing a persistent state, and which therefore could be shared with impunity between distributed Ada software, was termed a template unit. The concept therefore also admits all generic units and subprograms without side effects.

Although "template unit" is perhaps not the best name, particularly for library subprograms, it is retained in DRAGOON for continuity. Moreover, DRAGOON also permits the use of all other library units conforming to the rules for template units defined in DIADEM. This facilitates the direct reuse of large libraries of reusable subprograms, such as those of NAG, and also of the libraries of commercially developed "abstract data type" packages. Template packages in DRAGOON may contain the definition of types, subprograms, constants and exceptions, but not variables or tasks.

3.2.1 Templates in the Supermarket Control System

Since the type definitions used to represent product attributes need to be visible to a number of the classes in the system, they need to be defined in template packages which can be accessed by any class. The bar-code and stock-level attributes (section 1.4), being simple natural numbers, are clearly best represented by variables of the Ada type *NATURAL*. By storing prices in terms of pence, this also becomes the most straightforward representation of the price attribute. However, rather than use the type *NATURAL* directly, it is safer to define separate derived types for each, since this ensures that the attributes will never be mixed. The textual label, on the other hand, clearly needs to be represented as a character string. Since a string subtype in Ada must be of a certain specified length, a size of 100 is chosen arbitrarily.

These four types are defined together in a single template package :-

```
package PRODUCT is                                                    /3.2/
    type CODE_NUMBER is NATURAL;
    type VALUE is new NATURAL;
    type STOCK_NUMBER is new NATURAL;
    type LABEL is subtype STRING range (1 .. 100);
end PRODUCT;
```

In view of the above discussion, the "sell-by-date" attribute of perishable products will also be represented as values of an abstract data type, but not as a single number like the others. In fact, it is tempting to reuse the type definitions in the standard Ada package, *CALENDAR*, [DoD, 83] for this purpose, but this is not a legal DRAGOON program unit since it is not a template package. Although it contains no obvious variable declarations, *CALENDAR* has a state by virtue of the function, *CLOCK*, which returns the current time (including the date) as measured by the system clock. This facility is not required for the purposes of recording a product sell-by-date. Only three subprograms are really necessary; a constructor function, *MAKE_DATE* say; a function to check for the equality of two dates, which by virtue of Ada's overloading facilities may be denoted by the infix operator "="; and a function to determine whether one date is earlier than another, which may similarly be called "<". Another template package, therefore, is required which provides similar type definitions to *CALENDAR*, but without provision for the time, and without a *CLOCK* function :-

```
package SIMPLE_CALENDAR is                                            /3.3/

    type DATE is private;

    subtype YEAR_NUMBER is INTEGER range 1901 .. 2099;
    subtype MONTH_NUMBER is INTEGER range 1 .. 12;
```

```
    subtype DAY_NUMBER is INTEGER range 1 .. 31;
    function MAKE_DATE (YEAR : YEAR_NUMBER;
                        MONTH : MONTH_NUMBER; DAY : DAY_NUMBER) return DATE;
    function "=" (D1, D2 : DATE) return BOOLEAN;
    function "<" (D1, D2 : DATE) return BOOLEAN;
private
        :
            -- implementation dependent
        :
end SIMPLE_CALENDAR;
```

3.3 Objects and Classes

Having defined the Ada types used to represent the attributes of products, it is now possible to consider the product descriptors themselves, and how they would be modelled in DRAGOON as objects. As mentioned above this is achieved by the definition of a *class*, which is a "blue-print" describing the properties that all objects or *instances* of this class will possess. As such, a class can be regarded as an "object type".

DRAGOON classes consist of two separate parts - the *specification part* and the *body part*. These are separately compilable library units, and like their Ada package counterparts, the specification must be compiled before the body. Unlike tasks types in Ada, however, it is not possible to nest one class definition within another. *All* DRAGOON classes are library units, therefore.

The class specification defines the *interface* that instances of the class present to other objects, while the body defines their *implementation*, and hides all aspects which should not be externally visible. Since the private variables of an object should not be directly manipulated by other objects, these must be defined in the body of the class. DRAGOON, therefore, allows only *methods* and *exceptions* to appear in class interfaces. This means that specific *access* functions must be defined to determine the values of internal state variables, and *update* operations to change their values.

In the case of the objects representing product descriptors, therefore, methods are required for accessing and updating all four attributes. In addition, a method, *DECREMENT_STOCK_NUMBER*, will be provided for the use of checkout points so that when items are purchased and removed from the store they can decrement the stock level recorded in the corresponding product descriptor. The specification of a DRAGOON class defining product descriptors is thus of the form :-

```
with PRODUCT;                                                    /3.4/
class PRODUCT_DESCRIPTOR introduces
    function PRICE return PRODUCT.VALUE;
    procedure SET_PRICE_TO (PRICE : in PRODUCT.VALUE);
    function BAR_CODE return PRODUCT.CODE_NUMBER;
    procedure SET_BAR_CODE_TO (BAR_CODE : in PRODUCT.CODE_NUMBER);
    function STOCK_LEVEL return PRODUCT.STOCK_NUMBER;
    procedure DECREMENT_STOCK_NUMBER;
    procedure SET_STOCK_TO (STOCK_LEVEL : in PRODUCT.STOCK_NUMBER);
    function LABEL return PRODUCT.LABEL;
    procedure SET_LABEL_TO (LABEL : in PRODUCT.LABEL);
end PRODUCT_DESCRIPTOR;
```

This illustrates how closely the syntax of DRAGOON follows the style of Ada. The "with" clause is used to give the class visibility of the type definitions in the template package *PRODUCT* (3.2). DRAGOON does not allow use clauses to be used in classes, however, although they can be used as normal in template packages.

The method specifications in the interface define the *protocol* of the class, and represent the operations that instances of the class can perform. They have precisely the same syntactic form as Ada subprograms, and differ semantically only in that *conceptually* new copies of the methods are generated along with each new instance of the class.

3.3.1 Clientship

There are two principal mechanisms by which DRAGOON classes can depend on each other. One is by the *inheritance* mechanism mentioned in the last chapter and discussed further below. The other is by *clientship*, in which one class, the *client* is implemented using the services offered by an instance of another, the *server*. As far as clients are concerned, a server class is completely defined by the interface exported in the specification part, since this defines the set of methods that instances of the class offer for invocation. Once the specification of a class has been compiled, therefore, it is possible to define client classes which employ the services of its instances. The way in which classes are implemented in terms of others through clientship determines the communication pattern of their instances at run-time. Clientship, therefore, is also a relationship between objects, defining a *dependency* hierarchy which may vary dynamically according to which instances of a class are serving a particular client.

To generate instances of a class it is necessary to declare an *instance variable* of

the appropriate type. An instance variable is analogous to an access variable[3] in Ada in that it stores an indirect reference to (or internal name for) a dynamically generated entity. Whereas normal access variables point to data structures, instance variables always refer to class instances - that is, objects. In this context, therefore, the adjective "instance" is not used in the sense of Smalltalk to distinguish variables in a class object (class variables) from those in instances (instance variables) or in methods (temporary variables), but merely to distinguish variables pointing to class instances from those that denote conventional data structures. As mentioned above, DRAGOON permits the use of both static and dynamically allocated data types, as well as classes.

A new object is brought into existence by invoking the special predefined operation, *CREATE*, through an instance variable. It is convenient to think of this operation as a method [Meyer, 88], since it is invoked in the same manner as other methods, but strictly speaking it is not a method because no object exists to receive and execute the *CREATE* request before it is performed. In fact, in pure object-oriented languages such as Smalltalk, in which classes are regarded also as objects with a run-time existence, the *CREATE* operation is elegantly handled as a method of the class object. In DRAGOON, which has a static view of classes, it is more closely analogous to the "*new*" operation of Ada which generates dynamically-allocated data structures. However, provided that the very special nature of the *CREATE* operation is recognised, there is no difficulty in referring to it as a method. In particular, it is important to note that although every class possesses a *CREATE* "method" (except special abstract classes described in (section 4.3)) it is not part of the protocol of instances.

Consider the class modelling the central control point in the supermarket control system, for example. This is responsible for generating new product descriptor objects when new product types are introduced into the supermarket. To achieve this, an instance variable of class *PRODUCT_DESCRIPTOR* (3.4) would be defined in the body of the class :-

NEW_DESCRIPTOR : *PRODUCT_DESCRIPTOR*;

Elaboration of this declaration does not bring a new object into existence, but merely an instance variable capable of holding a reference to one. Until an existing object is assigned to it, or a new one generated, the value of the reference denoted by this variable is "**null**", just as with newly elaborated access variables in Ada.

As in Eiffel and Trellis/Owl, the invocation of a method in DRAGOON has a similar form to a subprogram call, but is prefixed, in the Ada dot notation style,

[3]Strictly speaking an access *object* in Ada, but this term is avoid in DRAGOON because of the clash with object-oriented terminology.

with the name of the object to which the invocation request is directed. Therefore, the invocation of the *CREATE* "method" bringing an object into existence has the form :-

NEW_DESCRIPTOR.CREATE;

and causes the instance variable to refer to a new instance of class *PRODUCT_DESCRIPTOR*. Similarly, the price attribute of the object just created can be set to the value 50, say, by invoking its *SET_PRICE_TO* method :-

NEW_DESCRIPTOR.SET_PRICE_TO (50);

If the method invoked is a function, then the method call is an expression, otherwise it is a statement. Invocation of a method in this style corresponds to the transmission of a message in Smalltalk terminology.

3.3.2 Class Bodies

The clients of a class can be defined and compiled as soon as the specification is available, but as with Ada packages, nothing can actually be executed until the body part is also compiled. Each method defined in the class specification (except those that are deferred (section 4.3)) must be implemented in the body of the class. The body is also responsible for declaring the variables that hold the state of instances of the class. In the case of the *PRODUCT_DESCRIPTOR* objects, four variables are needed to store their state :-

```
class body PRODUCT_DESCRIPTOR is                                    /3.5/
    INTERNAL_BAR_CODE : PRODUCT.CODE_NUMBER;
    INTERNAL_PRICE : PRODUCT.VALUE;
    INTERNAL_STOCK_CODE : PRODUCT.CODE_NUMBER;
    INTERNAL_LABEL : PRODUCT.LABEL;
    function PRICE return PRODUCT.VALUE is
    begin
        return INTERNAL_PRICE;
    end;
    procedure SET_PRICE_TO (PRICE : in PRODUCT.VALUE) is
    begin
        INTERNAL_PRICE := PRICE;
    end;
        :
            -- similarly for other methods
        :
end PRODUCT_DESCRIPTOR;
```

Only two method bodies are illustrated in this example since the others are also simple accessor and update functions implemented in an analogous fashion. Not only do methods have the same syntactic form as Ada subprogram bodies, but within the class body they have the same semantics, since the visibility rules are identical to those of a package body; the variables declared in the body being directly visible in the following method bodies. Whereas there can only be one instance of a package, however, it is possible to generate many instances of a class. Conceptually each new instance possesses a separate copy of the class body, so that the methods, as well as the state variables upon which they operate, are replicated. In this respect classes are very similar to Ada generics; the only difference being that generics have to be instantiated at compile time with a fixed name, whereas classes can be instantiated dynamically with an indirect reference. As with Ada generics, although "methods" are conceptually replicated for each instance of a class, the code for their implementation is shared.

It is possible to define subprograms in the body of a class which do not appear in the specification. These can be thought of as *internal methods*, similar to other methods in every way except for the fact that they are not made visible in the interface. However, since they are essentially nothing more than Ada subprograms defined internally, it is convenient in DRAGOON to refer to them as such, and to reserve the word "method" for those subprograms which are exported in the interface. Thus, within the body of a class internal operations of this kind can be invoked as normal Ada subprograms instead of using some clumsy notation for introspective method invocations such as the pseudo-variable, *SELF*, of Smalltalk.

DRAGOON does not permit the (externally visible) methods of a class to be invoked directly from within the body, since this can easily lead to erroneous programs, particularly in a concurrent environment (Chapter 5). The same effect can easily be achieved by defining an internal subprogram to implement the body of the method (i.e. arranging for the method to invoke the internal procedure), which can be called from within the body of the class.

3.4 Inheritance

The last few sections have described the facilities available in DRAGOON for defining completely new classes. However, the inheritance mechanism of object-oriented languages such as DRAGOON also makes it possible to define new classes simply by specifying how they differ from existing ones. In other words, a class can *inherit* the properties of another, and extend or adapt them to meet new requirements. The inheriting class, or the *heir*, therefore, becomes a specialisation of the inherited

class, or *parent.*

In the case of the supermarket control system, for example, certain categories of products have additional attributes - perishable products have an extra "sell-by-date", while electrical products have an extra "fuse rating". In a conventional language such as Ada, a new descriptor type would have to be defined for such products, involving the redefinition of all the common methods, not just those that are effected by the change. In DRAGOON, on the other hand, it is only necessary to define an heir of *PRODUCT_DESCRIPTOR* (3.4) which inherits all the common attributes and introduces the additional ones needed. Thus, a specialisation of the class *PRODUCT_DESCRIPTOR* to represent perishable items would have the following form :-

```
with SIMPLE_CALENDAR;                                              /3.6/
class PERISHABLE_DESCRIPTOR
    inherits PRODUCT_DESCRIPTOR;
    introduces
        function SELL_BY_DATE return SIMPLE_CALENDAR.DATE;
        procedure SET_SELL_BY_DATE_TO (SELL_BY_DATE : in
                                        SIMPLE_CALENDAR.DATE);
end PERISHABLE_DESCRIPTOR;
```

This indicates why the keyword "introduces" appears before the list of new methods. The methods *SELL_BY_DATE* and *SET_SELL_BY_DATE* are introduced (i.e. added) to the set of methods offered by instances of class *PRODUCT_DESCRIPTOR* (3.4). The protocol of *PERISHABLE_DESCRIPTOR* objects is thus a union of that inherited from the parent, and that "introduced" by the class specification. Since the parent class may, in turn, have inherited methods from its parent, in general, the protocol of a class is the union of those of all its ancestors in the inheritance hierarchy. The inheritance relation is transitive, therefore, so that if a class X inherits from another Y, and Y inherits from Z, then X also inherits the properties of Z. X is not said to be the heir of Z, however, or Z the parent of X, because these terms are reserved only for the classes immediately involved in the inheritance relationship. In other words, the parent(s) of a class are those appearing in its inherits clause, while the heirs of a class are those that name it in their inheritance clauses. To describe classes indirectly related by inheritance the terms *ancestor* and *descendent* are introduced. The ancestors of a class are all those classes from which it directly or indirectly inherit its features, while the descendents are all those classes which directly or indirectly inherit its features.

Although the specification of an heir class can be compiled before the body of its parent the same is not true for its body. Inheritance is essentially a mechanism for sharing the implementation of classes, and unlike the clientship relationship,

therefore, in general involves the heir class having access to the "hidden" information in the bodies. Consequently, the code in the body of a class is not hidden from its heirs as it is from its clients.

The body of *PERISHABLE_DESCRIPTOR* is of the form :-

```
class body PERISHABLE_DESCRIPTOR is                                    /3.7/
    INTERNAL_DATE : SIMPLE_CALENDAR.DATE;
    function SELL_BY_DATE return SIMPLE_CALENDAR.DATE is
    begin
        return INTERNAL_DATE;
    end;
    procedure SET_SELL_BY_DATE_TO (SELL_BY_DATE : in
                                        SIMPLE_CALENDAR.DATE) is
    begin
        INTERNAL_PRICE := SELL_BY_DATE;
    end;
end PERISHABLE_DESCRIPTOR;
```

In the same way that new methods "introduced" in the specification extend the interface of class instances, the variables defined in the body extend their set of state variables. *PERISHABLE_DESCRIPTOR* objects, therefore, have five variables, four inherited from *PRODUCT_DESCRIPTOR* (3.4) and one declared directly in the class body. Although in this case the methods introduced by *PERISHABLE_DESCRIPTOR* do not have cause to manipulate the inherited variables, they could easily have done so. All declarations (including types, constants and variables) contained in the bodies of ancestors of a class are in scope in its body. Any method implementation, therefore, may reference inherited variables directly, as if they had been defined explicitly in the same body part. This implies also that "with" clauses are inherited. It is not possible, however, to invoke the inherited methods from within the bodies of introduced methods, because as mentioned above, no class is allowed to call its own methods directly.

A class can be both a descendent and client of another class. Consider the descriptor for electrical products, for example. As mentioned in chapter 1, rather than simply record the associated fuse rating as a numerical value, this information is stored by reference to the product descriptor for the appropriate fuse. Just like *PERISHABLE_DESCRIPTOR*, therefore, electrical product descriptors need to be an heir of the class *PRODUCT_DESCRIPTOR* (3.4), but they also need to be a client so that they may contain a reference to the product descriptor for the appropriate fuse :-

```
with PRODUCT_DESCRIPTOR;                                          /3.8/
class ELECTRICAL_DESCRIPTOR
    inherits PRODUCT_DESCRIPTOR;
    introduces
        function REQUIRED_FUSE return PRODUCT_DESCRIPTOR;
        procedure SET_FUSE_TO (FUSE : in PRODUCT_DESCRIPTOR);
end ELECTRICAL_DESCRIPTOR;


class body ELECTRICAL_DESCRIPTOR is
    INTERNAL_FUSE : PRODUCT_DESCRIPTOR;
    function REQUIRED_FUSE return PRODUCT_DESCRIPTOR is
    begin
        return INTERNAL_FUSE;
    end;
    procedure SET_FUSE_TO (FUSE : in PRODUCT_DESCRIPTOR) is
    begin
        INTERNAL_FUSE := FUSE;
    end;
end ELECTRICAL_DESCRIPTOR;
```

All the previous classes considered have had only variables of static data types. This class, however, also possesses an instance variable of the class *PRODUCT_DESCRIPTOR* (3.4), containing a reference to an instance of the class, rather than a data value. In general, there are three different kinds of variables (sometimes also called *fields*) which may appear in the body of a class; instance variables for holding references to objects (i.e. class instances), access variables holding references to dynamically generated data structures (but not objects) and *data* variables for holding values of *static* data types. In DRAGOON variables of the last kind, and/or the values they contain, are termed the *attributes* of the object. This is a different usage of the term to that found in Eiffel and some other languages, where it is the instances variables which are referred to as attributes. The DRAGOON interpretation, however, corresponds more closely to the intuitive meaning of the word, because as discussed in section 3.1 (static) data types are used to model qualities (i.e. attributes) of objects, rather than objects themselves. In DRAGOON, therefore, the attributes of a class are its variable *qualities*, not the objects of which it may be a client, or the variables referring to them.

3.4.1 Method Modification

In the examples of inheritance above, the heir classes have merely extended their parents by adding new properties to those they inherited. Specialisation, therefore,

has been achieved by *enrichment.* However, it is also possible to specialise a class by *modification* of the inherited properties. More specifically, it is possible to modify the methods inherited in the following ways :-

Method Redefinition. Probably one of the most important ways in which an heir can modify the inherited properties is by *redefining* one or more of the inherited methods. DRAGOON permits an heir class to replace the *body* of an inherited method by a new implementation, but does not allow the specification of the method to be changed. "Reimplementation", therefore, is probably a more accurate description of this facility. In circumstances where the "redefinition" of a method involves merely extending the inherited definition with extra steps immediately before and after, it is possible to "reuse" the inherited version in the definition of the new one.

Some languages, such a Eiffel, permit the types of the parameters to be changed in the redefinition of a method, but this has serious consequences for the typing scheme of the language, as discussed in the next section. Since DRAGOON does not permit such redefinition, and the specifications of inherited methods remain unchanged, the reimplementation of inherited methods is made known to clients by means of a "**redefines**" clause in the heir's interface. The specification of the following class, for example, indicates that it redefines the method *SELL_BY_DATE*, and in its body, therefore, provides a new implementation (i.e. body) for the method :-

```
class REDEFINED_DESCRIPTOR                                    /3.9/
    inherits PERISHABLE_DESCRIPTOR;
    redefines SELL_BY_DATE;
end;
```

It is important to note that reimplementation can be used for two subtly different purposes. The new version of a method defined in an heir can stay faithful to the semantics of the original version and merely adopt an alternative implementation strategy, or alternatively, can provide a different service to the original, perhaps bearing no relationship whatsoever. The second case gives rise to a property of object-oriented languages known as dynamic binding, explained in the next chapter.

Method Renaming. As well as being able to redefine the body of an inherited method, it is also possible to change the name associated with it. This is achieved using a "**renames**" clause in the specification, which indicates to a client that the specified inherited method is to be known by a new name. In the case of single inheritance this is merely a convenient facility for improving program understandability, but in the case of multiple inheritance, introduced below, it is sometimes essential for overcoming name clashes. It is, in fact, possible both to rename and redefine an inherited method.

The renaming of methods in the specification of a class does not actually alter the interfaces of its instances, but rather the set of methods callable through instances variables of the class. Consider the following heir of *PERISHABLE_DESCRIPTOR*, for example :-

```
class RENAMED_DESCRIPTOR                                          /3.10/
    inherits PERISHABLE_DESCRIPTOR;
    function PERISHED_BY return SIMPLE_CALENDAR.DATE renames SELL_BY_DATE;
end;
```

An attempt to call this renamed method via an instance variable of class *RENAMED_DESCRIPTOR* using the old name (*SELL_BY_DATE*) will be rejected by the compiler, since it can only be called from such an instance variable using the new name (*PERISHED_BY*). If the object concerned is assigned to an instance variable of class *PERISHABLE_DESCRIPTOR* (3.6), however, the reverse is true. From such an instance variable the method is known by its old name. Renaming, therefore, effects the interfaces associated with instance variables, rather than objects.

Method Restriction. Perhaps the most drastic modification that can be made to inherited methods is their complete removal from the interface of the heir. This is sometimes necessary when the heir models a specialisation of the parent in which it would be an error to call certain of its methods. It is achieved by a special "removes " clause in the specification of a class similar to those for renaming and redefines :-

```
class RESTRICTED_DESCRIPTOR                                       /3.11/
    inherits PERISHABLE_DESCRIPTOR;
    removes SELL_BY_DATE;
end;
```

These method modification facilities, and the different ways in which they can be defined, have an important part to play in the use of DRAGOON for concurrent and distributed systems, and the power of object-oriented languages generally for promoting reuse, but also have an important impact on the typing scheme of the language. These issues will be discussed as they arise in the following chapters.

3.4.2 Multiple Inheritance

Another important extension of the basic inheritance mechanism introduced above is multiple inheritance - the possibility for a class to inherit the properties of more than one parent. This is exhibited by a number of languages, including Eiffel [Meyer, 88] and Flavors [Moon, 86]. Suppose, for example, that a class *DATED_ITEM* were available in the DRAGOON library which simply possessed an attribute of type

SIMPLE_CALENDAR.DATE (3.3) and associated accessor and update methods. When instantiated directly, such a class, with a specification of the form :-

```
with SIMPLE_CALENDAR ;                                          /3.12/
class DATED_ITEM introduces
    function DATE return SIMPLE_CALENDAR.DATE;
    procedure SET_DATE_TO (DATE : in SIMPLE_CALENDAR.DATE);
end DATED_DESCRIPTOR;
```

would be of little value, since instances would simply encapsulate a single "date" attribute, and in most circumstances, therefore, would be no more useful than the attribute itself. However, when combined with other objects using multiple inheritance the class is extremely useful, since it enables any kind of object which possess a date attribute (but others also) to be modelled cleanly and efficiently. There are many examples of such classes, such as people with birthdays, students with exam dates, and products with sell-by dates.

Given the availability of *DATED_ITEM*, a class with identical properties to *PERISHABLE_DESCRIPTOR* (3.6) could have been generated from *PRODUCT_DESCRIPTOR* (3.4) simply by also inheriting the properties of *DATED_ITEM* (3.12) rather than by explicitly introducing extra methods and attributes :-

```
class DATED_DESCRIPTOR                                          /3.13/
    inherits PRODUCT_DESCRIPTOR, DATED_ITEM;
end DATED_DESCRIPTOR;
```

Including the effort required to define the *DATED_ITEM* (3.12) class in the first place, there is little difference to choose between the *PERISHABLE_DESCRIPTOR* "route" for generating "perishable-food" descriptors, and the *DATED_DESCRIPTOR* (3.7) "route" using multiple inheritance. The point is, however, that the properties of *DATED_ITEM* can also be inherited by any other class which needs to have an additional date attribute.

In a simulation of a library, for example, classes will be required to model the books handled by the library. Independently of its status in a library, a book is a well defined abstraction with fixed attributes such as an author, title, publisher etc., and so it is likely that a class will be defined, *LIBRARY_BOOK* say, to model the basic book abstraction. As soon as a book is borrowed from the library, however, it assumes an extra set of attributes, one of which will be the return date (others possibilities are the borrowers name, address etc.). Using multiple inheritance, the required date attribute can be added to the new version of the class simply by inheriting the class *DATED_ITEM* along with *LIBRARY_BOOK* :-

```
class DATED_LIBRARY_BOOK                                    /3.14/
    inherits LIBRARY_BOOK, DATED_ITEM;
end DATED_LIBRARY_BOOK;
```

Multiple inheritance, therefore, provides a mechanism for factorising common information into inherited classes, and greatly increases the range of logical relationships that can exist between classes. Whereas single inheritance enables objects to be arranged into subsets, instances of an heir class belonging to a subset of those of the parent, multiple inheritance permits them to be arranged into set inter-sections. Instances of *DATED_LIBRARY_BOOK* (3.14) for example belong to the intersection of the set of *DATED_ITEM* (3.12) instances and *LIBRARY_BOOK* instances.

In order to overcome the name clashes which arise when the parents of a class have identical method specifications or variable names, DRAGOON permits the inherited identifiers to be qualified by the name of the parent class in the Ada "dot-notation" style. When the clashing identifiers are declared in the body of the parent classes, this is sufficient to overcome any ambiguities. If, for example, the *LIBRARY_BOOK* and *DATED_ITEM* parents of *DATED_LIBRARY_BOOK* each had a variable called *COUNTER*, code in the body of *DATED_LIBRARY_BOOK* could distinguish these variables by using the identifiers *LIBRARY_BOOK.COUNTER* and *DATED_ITEM.COUNTER* respectively.

When the clashing identifiers appear in the specifications of the parents, however, this technique alone is not sufficient, since clients of a class should not require knowledge of the ancestry of a class in order to unambiguously call methods of its instances. In such circumstances, the programmer is required to use the renaming facilities of the language to rename one, or both, of the clashing methods.

3.5 DRAGOON Typing Scheme

Most languages intended for the development of commercial and industrial software employ some form of typing scheme to help minimise logical errors and improve program reliability. This shortens the debugging stage of program development, which is often one of the most costly parts, and also reduces the likelihood of undetected software errors occurring at a later stage when the software has been supplied to the client. These are particularly important requirements in the field of embedded systems where software often plays a critical, controlling role, and the consequences of a software failure could be disastrous. Languages, such as Ada, which are intended primarily for this application domain therefore usually adopt a *strong typing* scheme in which *every* value in a program has an associated type, and all operations are checked to ensure the logical compatibility of the operands.

It is possible to perform type checking dynamically by tagging each data structure with a type label, and checking that the labels of operands are consistent before operations are performed[4]. Dynamic typing of this form, however, carries unacceptably high run-time overheads for real-time applications. It is preferable in such applications for as much of the type checking as possible to be performed *statically* by the compiler before the program is executed.

Most statically typed languages require the programmer to provide supplementary type declarations which explicitly define the type of every data structure employed in the program. Much of this information is actually redundant, as demonstrated by languages such as ML [Milner, 84] which ensure the type-consistency of programs with only minimal type declarations. Nevertheless, providing explicit types for every operand significantly reduces the work that has to be carried out by the compiler, and improves the readability and correctness of source code.

The benefits of static typing have to be carefully balanced against the constraints on the flexibility of the language. This is particularly so in object-oriented languages, one of whose most important features is the polymorphism made possible by the reference semantics used to identify objects. Since objects may be assigned to any instance variable in languages like Smalltalk, a method call may be directed to, and serviced by, many different objects during the course of a program. This property is largely responsible for Smalltalk's power for rapid prototyping, but at the same time accounts for most of the run-time errors that occur due to inappropriate assignments.

Too rigid a typing scheme also directly conflicts with the interests of reuse because it restricts a program component to operating on exactly one set of types. This often makes it necessary to define components which are identical in every way apart from the types they operate on, even though these types may have identical representations in the computer. Even Ada, therefore, has a number of mechanisms for ameliorating the undesirable restrictions of strong typing. Ada's typing scheme is not completely static, but allows the programmer to define *subtypes* by specifying dynamic constraints on the values a variable may assume. The set of values defined by a subtype is a proper subset of that of the type itself. In the case of derived types, Ada also permits the programmer to explicitly convert one type to another. Finally, Ada permits components to be parameterised with respect to types using generics.

As DRAGOON is intended to be an enhancement of Ada in its intended domain of embedded systems, DRAGOON shares the same basic goal of using strong typing to improve program correctness and reliability. Like other object-oriented

[4]In fact, variant records in languages like Ada and Pascal are just such structures, with the discriminant acting as a dynamically-checkable type label

languages with strong typing, however, such as Emerald, Trellis/Owl and Eiffel, DRAGOON aims to strike a fine balance between the safety advantages of typing, and the flexibility and reuse advantages of object-oriented languages, particularly polymorphism.

3.5.1 Class and Data Types

DRAGOON differs from the majority of object-oriented languages, in that it deliberately incorporates the full power of Ada's typing and encapsulation mechanisms. In discussing the typing rules, therefore, it is important to distinguish carefully between the various possible representations of data in the language.

A "type" is associated with every entity in the program upon which operations may be performed, including simple assignments from one variable to another. Given the "hybrid" nature of DRAGOON, therefore, there are two different kinds of entities to which types may be attributed; objects, whose types are defined by the classes of which they are an instances, and data elements, whose types are defined by type declarations in the traditional Ada style. In other words, there are two different kinds of type in DRAGOON; *class types* and *data types.* Every DRAGOON class, therefore, is a type, but not every type is a class.

Following the normal Ada terminology, it is convenient to further subdivide data types into two different kinds; *static* data types which are allocated on the stack as part of the elaboration of the program unit in which they are defined, and *dynamic* data types which are allocated on the heap whenever the corresponding allocator operation is invoked. Dynamic data types behave in many ways similar to class types in that elements of the type may be generated dynamically at any point in the program, and are referenced indirectly. Nevertheless it is important to be aware of the conceptual difference between them. An element of a dynamic type is merely a data structure referenced indirectly by an access variables, whereas an instance of a class is an object, encapsulating both data and associated operations. These principal categories of types in DRAGOON are illustrated in Figure 3.3.

Distinguishing between these different types in DRAGOON reinforces the terminology introduced above to distinguish between the two possible representations of "data abstractions" as "abstract data types" and "classes". Pure data abstractions in DRAGOON may, thus, be represented in either one of the two principal categories of types. Similarly, the term "instance variable" distinguishes variables intended to refer to *instances* of a class type (i.e. objects) from "data variables" or "attributes" intended to contains values of a data type.

Because classes are normal types in the languages, they can be used in the construction of compound data structures. It is possible for the elements of an

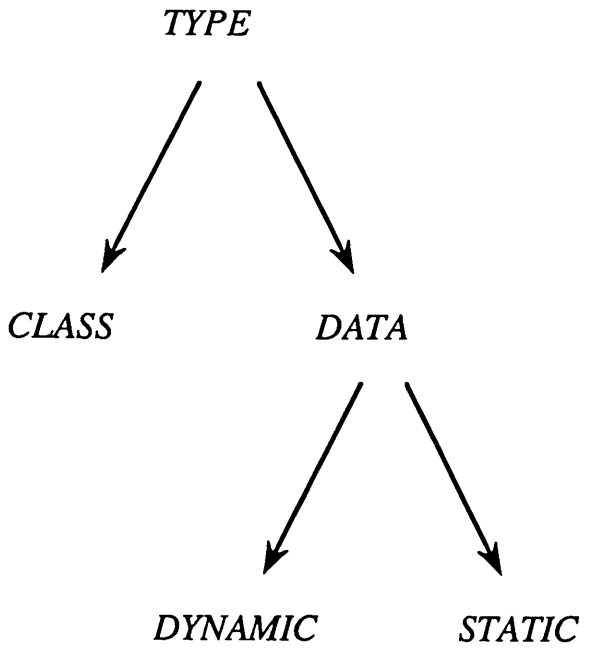


Figure 3.3: Type Categories in DRAGOON

array and the fields of a record to be of a class type (i.e. objects). Classes, therefore, are fully integrated into the typing rules "inherited" from Ada. Naturally, as far as data types are concerned, the associated typing rules are precisely those of Ada. Similarly, the rules for compound data structures are the same as those of Ada except in so far as the elements or fields are objects of class types. In this case, it is the rules related to class types discussed below which apply.

3.5.2 Type Compatibility

The basic goal of typing is to ensure that every entity in a program is used in a manner consistent with its declared type. In the context of object-oriented programming, in which client objects make use of server objects by the invoking their methods, this transposes into the goal of ensuring that every attempted method invocation is meaningful for the object to which it is directed. In other words, static typing in object-oriented languages is intended to ensure at compile time that no attempted method invocation fails because the method is not part of the server's interface.

The methods of an object are actually invoked by issuing a method call through a reference (or internal names) stored in either :-

- an instance variable, or
- a formal method parameter.

which are *statically* declared to be of a particular class type. During the execution of the program, however, references to different types of objects may be assigned to these enties - in the first case by means of direct assignment statements, and in the second when actual parameters are mapped onto formal parameters during a method call. Without any constraints on these assignments, therefore, the class type statically associated with an instance variable or formal method parameter - its *static* type [Meyer, 88] - may not be the same as the class type of the object to which it is dynamically made to refer - its *dynamic* type.

To protect against the invocation of unrecognised methods, therefore, it is necessary to define typing rules which ensure that no instance variable or formal method parameter is ever made to refer to an object which is not able to respond to all the method calls that instances of the *static* class type can. Moreover, if these rules are to be statically checkable from the program text, it is necessary to check the compatibility of classes at the point of assignment, rather than at the point when the method call is made. In other words the type of the *source* of an assignment must be able to respond to at least all the method invocations (messages) to which the static type of the *target* (i.e. an instance variable or formal method parameter) may respond.

If the typing rules of the language permit an instance of a class S to be assigned to an instance variable or formal method parameter of a class T, then S is said to be *type compatible* with, or a *subtype* of, T. Type-compatibility is a transitive relationship, so that if S is a subtype of T, and T is a subtype of U, then S is a subtype (i.e. type-compatible) with U.

3.5.3 Conformance

The simplest way of deciding whether two classes are type-compatible is by the set of methods exported in their interfaces - that is, their *protocols*. If the interface of the source class S indicates that its instances are able to respond to all method calls that instances of the target class T can, then, adopting the terminology of Emerald [Black et al., 86] which was one of the first languages to adopt this approach, S is said to *conform* to T. Conformance, therefore, is not in anyway related to the implementation of classes, but is determined solely by their protocols.

Informally, for a source class S to conform to a target class T, S must export at least all the methods of T, and the (formal) results (i.e. the "out" parameters) of any methods in S common to T must conform to the corresponding argument in T. On the other hand, the (formal) arguments (i.e. the "in" parameters) of the methods of T common to S must conform to the corresponding arguments in S. Not only is this definition recursive, therefore, but the last condition (known as

contravariance) requires that conformance of the arguments holds in the opposite direction to that of the classes. This is because instances of S may be treated as (i.e. sent the parameters expected by) instances of T, so that the protocols of the parameters must export at least all the methods expected by instances of S.

3.5.4 Subtypes

Conformance is sufficient to serve as the basis for type compatibility rules, since it guarantees that the target of a method call will always be able to respond in some way or another. However, because it is only based on the form of class interfaces, classes may conform which bear no logical relationship to one another what so ever. For this reason, languages such as DRAGOON and Eiffel which possess an inheritance mechanism have an additional requirement for one class to be type-compatible with another - it must also be its ancestor. Type compatibility is no longer determined by the form of the interface alone, therefore, but also by the relationship of the implementations. Although it is *necessary* for a class to be an ancestor of another class to be type-compatible with it, this is not *sufficient* because of the method modification possibilities. It is clearly also necessary that the heir class conforms to the parent.

Eiffel permits an heir class not only to change the body of inherited methods, but also to change the class types of their formal parameters. Fairly elaborate rules have therefore been defined in Eiffel to specify when classes are type compatible, but unfortunately they are not completely water-tight [Cook, 89]. There are, in fact, a number of situations in which Eiffel's type security may be breached. Firstly, Eiffel regards an heir which removes a method from the interface of a class as type-compatible with (i.e. a subtype of) the parent. This immediately leads to run-time errors when an attempt is made to invoke the removed method from an instance variable of the parent class. Secondly, the rules constraining how the parameters types of inherited methods may be changed do not express the contravariance requirement outlined above. Thirdly, Eiffel also has a facility, known as the "type redefinition rule", which permits the class type of inherited instance variables to be changed to a type-compatible class. Unfortunately, however, this features interacts unsafely with other features of the language so that run-time typing errors are possible in programs that conform to the typing rules [Cook, 89].

[Meyer and Nerson, 89], as well as Cook [89], have described how these problems may be overcome, but these involve adopting even more elaborate rules than those in the present version of Eiffel. Moreover, it is questionable how necessary the ability to redefine the parameter types of inherited methods really is, because this amounts to a *logical* change in the purpose of the method, rather than a mere reimplementation.

Heir classes which employ this facility, therefore, are not really logical subtypes of the parent, but rather alternatives. It is as if the method whose parameter types are redefined is removed from the interface, and a new method with the same name inserted.

To ensure the consistency of the typing scheme, therefore, DRAGOON adopts a much simpler strategy than Eiffel for defining when descendent classes are type-compatible with their parents. In DRAGOON,

- an heir class which removes from its interface one or more methods inherited from a parent is not a subtype of (i.e. type-compatible with) that parent.
- an heir class may only redefine the body of an inherited method; the interface must remain unchanged (except for parameters with default values (see below)).
- an heir class is *not* permitted to change the class type of inherited instance variables.

Although DRAGOON does not permit the types of existing method parameters to be changed in descendent classes, it does permit additional parameters to be added, provided they are given a default value. Like the corresponding Ada mechanism for subprogram parameters, when invoking a method it is not necessary to provide actual parameters for the formal parameters with default values, since the formal parameters take the value of the default if no other value is supplied. The redefinition of methods by adding default parameters does not destroy their ability to respond to method calls directed to instances of the parent class, therefore.

3.5.5 Terminology

Before proceeding further it is important to clarify the different ways in which the terminology associated with typing in object-oriented languages is used in DRAGOON. In some languages, such as Eiffel, the term conformant is taken as synonymous with "type-compatible". In other words, if a class S is conformant with a class T in Eiffel, then it is also type-compatible, and vice versa. This is not the case in DRAGOON, however, which adopts the original meaning as defined in Emerald [Black et al. 89]. As described above, a class S is conformant to another class T if its *interface* satisfies certain rules in relation to T's. This does not guarantee, however, that the typing rules will permit an instance of S to be assigned to an instance variable of class T, and hence it is not necessarily type-compatible.

Another source of ambiguity is the use of the terms subclass and superclass. Many object-oriented languages regard the term "subclass" as synonymous with

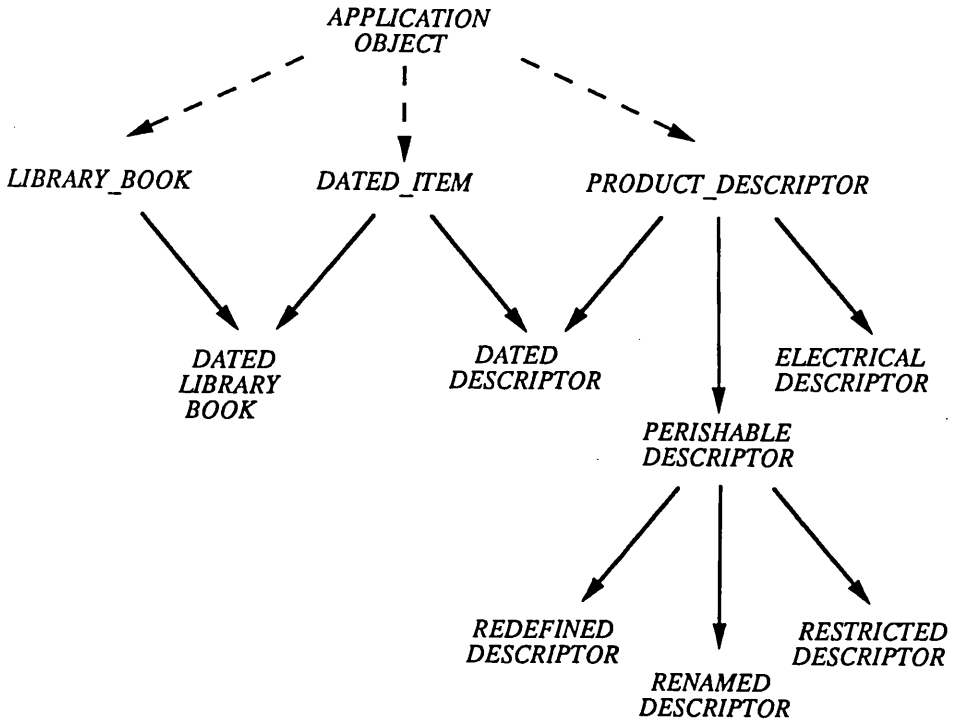


Figure 3.4: Inheritance Hierarchy

"heir" and "superclass" with "parent". In other words a subclass is an immediate descendent of its superclass. This terminology presents no problem in languages like Smalltalk which do not have a static typing scheme, but in languages which do, such as DRAGOON and Eiffel, it clashes unfortunately with the subtyping relationship between classes. If the term "subclass" were used synonymously with "heir" in DRAGOON, the typing rules outlined above would means that a sub-"*class*" would not always be a sub-"*type*", even though a class is supposed to be a type. There is, moreover, no need to use this terminology to describe inheritance relationships, because perfectly adequate terminology already exists ("heir/parent", "descendent/ancestor"). In DRAGOON, therefore, the term subclass is not regarded as synonymous with "heir" but rather with "subtype". A class S is a subclass of a type T if, and only if, it is a subtype - that is, type-compatible. Naturally, T is then termed a superclass of S.

Inheritance, subclassing and conformance in DRAGOON are therefore three closely related but distinct concepts. A subclass relationship implies both conformance and inheritance, but not vice versa. To illustrate these differences consider the various classes introduced so far during the course of this chapter, and illustrated in figure 3.4.

The class *PERISHABLE_DESCRIPTOR* (3.6), for example, conforms with *DATED-*

_ITEM (3.12), since it exports all its methods (i.e. *DATE*, *SET_DATE_TO*), but is not a subclass (i.e. type-compatible) because it is not an hier. Conversely, the class *RESTRICTED_DESCRIPTOR* (3.11) is an heir of *PERISHABLE_DESCRIPTOR* (3.6), but is not a subclass because it does conform with it. For a class to be a subclass of another it must be both an heir and conformant. *PERISHABLE_DESCRIPTOR* (3.6) is therefore a subclass of *PRODUCT_DESCRIPTOR* (3.4).

3.5.6 Type Conversion

The formulation of type compatibility rules such as those outlined above is a careful balancing act between the need to prohibit potentially dangerous assignments likely to cause run-time errors, and the need to allow a certain degree of flexibility in the structuring of programs. Nevertheless, however carefully the typing rules are designed, it is impossible to formulate purely static rules which will not be either too restrictive in some situations or too liberal in others. In other words, these two goals are essentially incompatible. Therefore, many languages with strong typing schemes, including Ada, provide mechanisms which the programmer may use to explicitly override the typing rules of the language in contexts where these are too restrictive.

Such facilities are also needed in DRAGOON because the static typing rules formulated above may, in certain circumstances, drastically undermine the usefulness of polymorphism for the purposes of reuse. To see why this is so, consider a class *STACK* which implements a stack of *PRODUCT_DESCRIPTOR* (3.4) objects :-

```
with PRODUCT_DESCRIPTOR;                                    /4.1/
class STACK introduces
    procedure PUSH (ITEM : in PRODUCT_DESCRIPTOR);
    function POP return PRODUCT_DESCRIPTOR;
end;
```

where *PUSH* and *POP* are the usual methods for adding and removing objects to and from the top of the stack respectively.

Since the classes *PERISHABLE_DESCRIPTOR* (3.6) and *ELECTRICAL_DESCRIPTOR* (3.8) are subclasses of *PRODUCT_DESCRIPTOR* (3.4) the typing rules of DRAGOON allow the programmer to take advantage of polymorphism and *PUSH* instances of both these subclasses onto a *STACK* object, as well as instances of *PRODUCT_DESCRIPTOR* itself. All the following *PUSH* operations are legal, therefore :-

```
    :
S : STACK
PRODUCT : PRODUCT_DESCRIPTOR
```

```
PERISHABLE : PERISHABLE_DESCRIPTOR
ELECTRICAL : ELECTRICAL_DESCRIPTOR
    :
    :
S.PUSH(PRODUCT);
S.PUSH(PERISHABLE);
S.PUSH(ELECTRICAL);
    :
```

The problem is that once instances of the subclasses of *PRODUCT_DESCRIPTOR* - *PERISHABLE_DESCRIPTOR* and *ELECTRICAL_DESCRIPTOR* - have been "pushed" onto *S* they lose their independent identity, and are essentially transformed into instances of *PERISHABLE_DESCRIPTOR*. Although the additional information associated with *PERISHABLE* and *ELECTRICAL* is stored in the stack, this can never be accessed again, because the static typing rules of DRAGOON forbid anything "popped" off *S* from being assigned to a subclass of *PRODUCT_DESCRIPTOR*. The following statements are therefore illegal :-

```
    :
PERISHABLE := S.POP;
ELECTRICAL := S.POP;
    :
```

even when the object on the top of *S* is, in fact, a *PERISHABLE_DESCRIPTOR* or *ELECTRICAL_DESCRIPTOR* object as appropriate.

This is a very serious constraint on the language's support for reuse, and a situation in which the static typing rules are clearly too restrictive. It is not possible to assign an object "popped" off *S* to a *PERISHABLE* instance variable, for example, even if the programmer ensures that nothing other than *PERISHABLE* objects are "pushed" onto it.

To overcome this unhappy state of affairs, DRAGOON provides a mechanism for ascertaining which class an object is an instance of, and for converting objects between type-compatible classes. So, for example, if the programmer was sure that the object on the top of *S* was an instance of *PERISHABLE*, he/she can explicitly convert the class type of the object returned by the *POP* method into *PERISHABLE* :-

```
    :
PERISHABLE := PERISHABLE'(S.POP);
    :
```

Naturally, the strategy used is very similar to that employed by Ada for explicit type conversions, the expression providing the value to be converted being prefixed by the name of the target type. The precise Ada convention of separating these with

a dot is not adopted, however, because it clashes with the DRAGOON notation for method calls.

If the programmer were not certain of the type of the object at the top of *S*, on the other hand, he/she may establish whether it is a particular subclass of the static (i.e. apparent) type using the "in" operation :-

```
    :
DESCRIPTOR := S.POP;
if DESCRIPTOR in PERISHABLE then
    PERISHABLE := PERISHABLE'(S.POP);
end if;
    :
```

Once again DRAGOON adopts the syntax of the corresponding operation in Ada - "in" is a binary operation returning true if the class of the object referenced by the left argument is a subclass of that of the right argument (in other words, if the dynamic type of the left argument is a subclass of the static type of the right). As in this illustration, this facility is most commonly used to check the class type of an object before an attempted type conversion. If a conversion is attempted in which the source is not a subclass of the target, a run-time error results.

These two facilities are extremely useful for overcoming the constraints of the static typing rules of the language in situations where they are too restrictive[5]. They are used entirely at the programmer's risk, however, since they also clearly permit the construction of erroneous programs that would otherwise be prohibited by the static typing rules.

[5]The omission of similar facilities from the current version of Eiffel is an unfortunate limitation on the languages support for reuse. The next version will have a facility for type conversion known as the "accept" instruction, [Meyer and Nerson, 89] but not for checking the type of objects.

Chapter 4

Software Reuse

Having introduced the basic features of DRAGOON, and illustrated how classes and objects are modelled in the language, this chapter focuses on the issue of software reuse, and describes why the inheritance, polymorphism and dynamic binding mechanisms of object-oriented languages such as DRAGOON provide such a powerful vehicle for this purpose. It then goes on to discuss how genericity, as found in Ada, compares with these mechanisms, and how it can be integrated with them in a natural way to provide complementary parameterisation opportunities to the programmer.

4.1 Component Engineering

As the costs of software development have soared over the last decade "reuse" has become a more and more important issue, and has spread to virtually all aspects of modern software engineering, including the reuse of designs, requirement specifications, documentation, testing material etc. Its ultimate goal, however, is to minimise the amount of new application software that has to be developed from scratch for each new system. Reusing design and requirements information enables consideration of reuse to be introduced at much earlier stages in the development process, thereby facilitating the reuse of larger bodies of software.

Traditional techniques for supporting reuse are based on the concept of what may be called "component engineering", in which new software systems are developed by assembling "reusable components" chosen from a large, carefully designed and tested library. This approach consequently divides the problems of promoting software reuse into two distinct considerations [Sommerville, 89a] - how to produce software components with maximum potential for reuse (i.e. design *for* reuse) and how to design new systems making the most effective use of such components (i.e. design *with* reuse).

Although it is fairly easy to state in general terms what attributes a "good" reusable component should possess (commonly accepted requirements include efficiency, simplicity, functional independence and maintainability [Booch, 87], [Matsumoto, 84]) it is much more difficult to define how one should go about designing components to exhibit these properties [Gautier et al., 89]. Object-oriented design is in fact one strongly favoured approach since it not only encourages good programming practice, but also leads to modular, "well structured" designs in which the object forms a natural unit of reuse. Many of the recent reuse methodologies targeted to Ada are consequently based on the ideas of object-oriented design [Booch, 87]

Modularity and independence alone, however, do not ensure that a component will be reusable. One of the most critical factors affecting "reusability" is the "generality" of a component, because this determines the range of different applications and contexts in which it may be employed. A component tailored specifically to one particular application is less likely to be reusable in different contexts than one designed with generality in mind.

The most important techniques for increasing the generality of components is *parameterisation* - each parameter that can be selected by the "reuser" at the time of reuse provides an extra degree of freedom that increases the component's range of potential uses. One of the oldest and simplest examples is the parameterisation of subprograms by data values, which was supported in the earliest high-level languages such as FORTRAN, and is the basis for the success of reusable mathematical routines (e.g. NAG). More recent languages have introduced new parameterisation facilities such as the parameterisation of components by types (e.g. genericity) and subprograms (e.g. procedure variables in Modula, generics in Ada). Genericity, in particular, is an extremely important tools for obviating the replication of essentially identical code in languages with strong typing, and will be discussed further below.

While increasing the parameterisation (i.e. the number of parameters) of a component may increase its "generality", however, it also significantly increases its complexity and the obscurity of its function. The "reusability" of a component is not necessarily maximised by identifying every conceivable operation that it may perform, and introducing as many parameter as possible. On the contrary, there is a practical limit to the extent to which a component can be parameterised without making it prohibitively complex and obtuse. A delicate balance needs to be achieved, therefore, between a component's *generality* and its *intelligibility* - a balance which is obviously harder to achieve for larger components with more scope for parameterisation.

4.1.1 Component Families

An important alternative to explicit parameterisation as a way of providing additional degrees of freedom in the use of a component is to define a "family" of distinct, but related components [1], each member representing either a variant of the basic component abstraction or providing a different implementation policy. Booch has been one of the main proponents of this approach for constructing reusable components, and has done much influential work on defining a detailed taxonomy.

All the principle "collection" components considered by Booch have at least two variants - the iterator and non-iterator - but most have quite a few more. An *iterator*, popularised in the language CLU [Liskov et al., 81], is a version of a component which provides operations enabling every encapsulated data item to be "visited" (i.e. have operations performed upon it) without destroying the structure itself. This facility is useful in a large variety of contexts where clients need to obtain information about the elements stored in a component without actually consuming any elements. A component may therefore be either an :-

- **iterator** - with iterator facilities included, or a
- **non-iterator** - with no such facilities.

Members of the family which differ only by their implementation policy rather than their interface are known as different *forms*. These, in turn, are divided into *spatial* and *temporal* characteristics. The spatial characteristics relate to the component's behaviour in a concurrent environment with multiple execution threads, and will be considered further in the next chapter. The spatial characteristics on the other hand relate to the component's memory requirements, and are therefore essentially orthogonal to the temporal ones. Components may either be :-

- **bounded** - with fixed or limited size, or
- **unbounded** - with an unlimited size.

Unbounded forms of a component must clearly be implemented using dynamic data structures (e.g. linked lists), whereas the bounded form can be implemented using either static or dynamic data structures.

The final spatial categorisation by Booch relates to the memory management policy of the component, which may either be :-

- **managed** - ensuring that redundant memory resources are reclaimed, or
- **unmanaged** - performing no explicit memory reclamation.

[1] In a sense this is just a special form of parameterisation with respect to the component name

Managed forms of a component take explicit steps to ensure that unused memory is reclaimed - that is, garbage collection is performed explicitly - while unmanaged forms provide no such guarantee.

Implementation in Ada

Although this taxonomy is essentially independent of any particular language, Booch primarily had Ada in mind during its development. Ada is undoubtedly better than most conventional languages for supporting this approach to reuse, since it supports modularity and information hiding by means of packages, concurrency in terms of tasks, and powerful parameterisation facilities in terms of generics. These facilities notwithstanding, it suffers from the same serious shortcoming as other conventional languages - each different variant and form of component has to be implemented independently as a separate package despite the fact that large parts of them may be identical. Not only does this vast replication of code significantly increase the effort required to produce the components in the first place, but it greatly increases the amount of space needed to store the components, the problems of organising and cataloguing them, and ultimately, therefore, the ease with which they can be retrieved and reused.

Consider, for example, the various different variations and forms of a simple "queue" abstraction. In its most abstract form, a queue is simply an ordered set of elements arranged according to the rule that elements can only be added at one end and removed from the other. In other words, queues impose a FIFO (first in first out) discipline on the addition and removal of elements from the set. As with other component families considered by Booch, queues can exist in all the different variants and forms mentioned above, that is iterator/non-iterator, bounded/unbounded and managed/unmanaged. In addition, Booch's identifies two other variants of the basic queue abstraction which allow element removal and addition to break the strict FIFO rules. A *balking* queue allows items to removed from an arbitrary position, while a *priority* queue allows items to be added to the queue at places other than the rear. With these additional variants the total number of meaningful versions of the queue abstraction is 104, each of which needs to be implemented separately as an independent package.

4.2 Component Hierarchies

The inheritance mechanism of object-oriented languages such as DRAGOON provides a much more elegant and efficient way of creating such a component family. Instead of representing the various possible forms of an abstraction as unrelated

members of a family of components, the inheritance mechanism allows the relationship between them to be modelled explicitly in an inheritance hierarchy. This not only greatly improves the organisation of information in the library, but also significantly reduces the work required to produce the family, since common parts of their implementation can be factored out into a single class which they all inherit.

This can be seen by considering how part of the queue component family could be represented in DRAGOON using the various facilities offered by inheritance. The problem of generalising the type of the elements in the queue (i.e. defining a queue component applicable to different types) will be considered further below. For the moment it is assumed that all the elements of the queue are instances of some, as yet unknown, class type, *ELEMENT*.

The following is a DRAGOON class representing the conceptually simplest variant of this component abstraction - the unbounded, unmanaged, noniterator variant :-

```
with ELEMENT;                                                    /4.1/
class SIMPLE_QUEUE introduces
    function POP return ELEMENT;          -- adopting Booch's "method" names
    procedure ADD (ITEM : in ELEMENT);
    function IS_EMPTY return BOOLEAN;
    UNDERFLOW : exception;
end SIMPLE_QUEUE;


class body SIMPLE_QUEUE is
    type QUEUE_POINTER;
    type QUEUE_NODE is record
        ITEM : ELEMENT;
        NEXT : QUEUE_POINTER;
    end record;
    type QUEUE_POINTER is access QUEUE_NODE;
    FRONT, LAST : QUEUE_POINTER;
    function POP return ELEMENT is
        FRONT_NODE : QUEUE_POINTER := FRONT;
    begin
        if FRONT = null then
            raise UNDERFLOW;
        else
            FRONT := FRONT.NEXT;
            return FRONT_NODE;
        end if;
    end;
    procedure ADD (ITEM : in ELEMENT) is
    begin
```

```
            if FRONT = :    then
                FRONT := new QUEUE_NODE;
                LAST := FRONT;
            else
                LAST.NEXT := new QUEUE_NODE;
                LAST := LAST.NEXT;
            end if;
            LAST.ITEM := ITEM;
        end;

        function IS_EMPTY return BOOLEAN is
        begin
            return FRONT = null;
        end;

    end SIMPLE_QUEUE;
```

It is an error to attempt to *POP* an element off an empty queue, so the class provides a boolean method *IS_EMPTY* to check whether an instance is empty, and defines the exception *UNDERFLOW* which is raised by *POP* if it is invoked under such circumstances. Since this class models an unbounded version of the queue abstraction, however, conceptually there is no limit to the number of items which can be added to the queue, and so no corresponding *IS_FULL* method and *OVERFLOW* exception is needed at this stage. The only practical constraint on the size of the queue is the amount of memory available on the heap. If this should be exhausted the exception *CONSTRAINT_ERROR* will be raised by the run-time system when the *ADD* method requests the allocation of a new *QUEUE_NODE*.

Notice how records and incomplete type definitions are used in the normal Ada style to define the dynamic "linked list". This fits in simply and naturally with the class/object concept, and so avoids the unnecessary complication of recursive class definitions which would have to be employed in other languages such as Eiffel and Smalltalk.

4.2.1 Iterator Queue

As mentioned above, one of the common variations on basic component abstractions is the inclusion of an iterator which enables each of the elements currently stored in a compound data structure to be "visited" - that is, to have some operation performed on them. Two ways of implementing iterators are considered by Booch: the passive form, in which the the client (or user) supplies the required operation as a parameter, and when so instructed the component ensures that it is performed on each element; and the active form, in which another object is provided (but as part of the overall component) which can access the elements of the queue and return them successively to the user for manipulation. The main difference is that in the

former the user does not gain access to the encapsulated data elements, but merely supplies the operation that needs to be performed upon them, while in the latter he/she does.

The implementation of the passive form in Ada depends on its powerful generic facilities which enable subprograms and packages to be parameterised with respect to other subprograms. This effect can be achieved in a fairly straightforward way in the object-oriented style as demonstrated below in section 4.4.4. However, it is in the simulation of the active form that the advantages of object-oriented programming and inheritance are most powerfully displayed.

In Booch's formulation an active iterator is a distinct "object" in the component with four characteristic operations :-

- **initialise** - prepare the component,
- **Get_Next** - advance the iterator to the next item,
- **Value_Of** - return the current item,
- **Is_Done** - return true if every item has been "visited".

Using inheritance, an iterator version of the component can be generated simply by creating a subclass of *SIMPLE_QUEUE* which adds the necessary operations to those already provided.

```
class SIMPLE_ITERATOR_QUEUE                                    /4.2/
    inherits SIMPLE_QUEUE;
    introduces
        procedure INITIALISE;
        procedure GET_NEXT;
        function VALUE_OF return ELEMENT;
        function IS_DONE return BOOLEAN;
        ITERATOR_ERROR : exception;
end SIMPLE_ITERATOR_QUEUE;


class body SIMPLE_ITERATOR_QUEUE is
    NEXT : QUEUE_POINTER;
    procedure INITIALISE is
    begin
        NEXT := FRONT;
    end;
    procedure GET_NEXT is
    begin
        if NEXT = LAST then
            raise ITERATOR_ERROR;
        else
```

```
            NEXT := NEXT.LAST;
        end if;
    end;

    function VALUE_OF return ELEMENT is
    begin
        return NEXT.ITEM;
    end;

        function IS_DONE return BOOLEAN is
    begin
        return NEXT = LAST;
    end;
end SIMPLE_ITERATOR_QUEUE;
```

In this class the iteration operations have been added to those of the *SIMPLE_QUEUE* (4.1), so there is no need for a separate "iterator" object as in Booch's implementation. This maintains the notion of classes as the unit of reuse, and has been achieved without reimplementation or modification of the existing *SIMPLE_QUEUE* code, as would be necessary in Ada.

4.2.2 Bounded Queue

In order to generate different *forms* of the abstraction, however, a certain amount of reimplementation is unavoidable, since by definition these embody different implementation policies. Nevertheless, inheritance permits the amount of reimplementation required to be restricted to those parts which are truly effected by the change. The following class, for example, represents a bounded version of the class *SIMPLE_QUEUE* (4.1) :-

```
class BOUNDED_SIMPLE_QUEUE                                    /4.3/
    inherits SIMPLE_QUEUE;
    redefines
        ADD, POP;
        CREATE (SIZE : in POSITIVE);
    introduces
        function IS_FULL return BOOLEAN;
        OVERFLOW : exception;
end BOUNDED_SIMPLE_QUEUE;


class body BOUNDED_SIMPLE_QUEUE is
    MAX_NUMBER, CURRENT_NUMBER : POSITIVE;
    procedure CREATE (SIZE : in POSITIVE) is
    begin
        MAX_NUMBER := SIZE;
```

```
        CURRENT_NUMBER := 0;
    end;

    function POP return ELEMENT is
        FRONT_NODE : QUEUE_POINTER := FRONT;
    begin
        SIMPLE_QUEUE'POP;
        CURRENT_NUMBER := CURRENT_NUMBER - 1;
    end;

    procedure ADD (ITEM : in ELEMENT) is
    begin
        if CURRENT_NUMBER < MAX_NUMBER then
            SIMPLE_QUEUE'ADD;
            CURRENT_NUMBER := CURRENT_NUMBER + 1;
        else
            raise OVERFLOW;
        end if;
    end;

    function IS_FULL return BOOLEAN is
    begin
        return CURRENT_NUMBER = MAX_NUMBER;
    end;

end BOUNDED_SIMPLE_QUEUE
```

This class illustrates the advantages of being able to redefine the *CREATE* "method" of an inherited class to specify new initialisation conditions. Although *BOUNDED_SIMPLE_QUEUE* implements queues with a fixed capacity[2], to increase the "generality" of the component it is clearly desirable for this number to be a parameter selectable by the reuser. This is achieved by redefining the class's *CREATE* method to receive the necessary arguments and perform the appropriate initialisation.

As pointed out in the previous chapter, the *CREATE* operation is the mechanism by which new objects are generated in DRAGOON, and although activated in the same way as a method, it is not really one. The "redefinition" of *CREATE*, therefore, conforms to different rules to that of normal methods, and is perhaps better thought of as *adding* initialisation actions performed at object creation time. Firstly, a "redefined" *CREATE* method may only have "in" parameters, since these may serve only as initialisation values. Secondly, a class inheriting from a parent whose *CREATE* operation has already been redefined must preserve the "inherited" parameters in any further redefinition. In other words, the redefinition of *CREATE* is cumulative, because it is essential that all initialisation values expected by the code inherited

[2]In fact, it is not a particularly efficient component, since although the size of the structure is bounded at any one time, it generates a new list node for each new element added to the queue. In other words, it is "unmanaged". Nevertheless, it serves as a useful vehicle for illustrating how inheritance can be used to add functionality to classes.

from the parent be supplied also by its descendents. Note, that since *CREATE* does not form part of the interface of a class, any redefinition does not effect the conformance of the class to other. In the current example, the *CREATE* operation is redefined to receive the required buffer size, and assign this value to the internal variable, *MAX_NUMBER* .

BOUNDED_SIMPLE_QUEUE also demonstrates how it is possible to reuse the "old", inherited version of a method in the redefinition of its body. The "new" versions of *POP* and *ADD* in *BOUNDED_SIMPLE_QUEUE* do not actually change the way in which the inherited operations are performed in the heir class, but merely add additional steps before and/or after the inherited part. Rather than replicate the code of the inherited method in the body of the redefined version, therefore, it is possible to incorporate the "old" version directly. For example, the old version of the method *POP* in *SIMPLE_QUEUE* is incorporated into the redefined version in *BOUNDED_SIMPLE_QUEUE* by the statement *SIMPLE_QUEUE'POP*. This should not be thought of as a method call, but as a direct substitution of the text of the parent method into the heir. Note, it is only possible to do this for the methods of heirs, not indirect ancestors.

4.3 Abstract Classes

Although use of the inheritance mechanism above enabled the class *BOUNDED_SIMPLE_QUEUE* (4.3) to be implemented more succinctly than would otherwise have been possible, it was still necessary to redefine most of the inherited methods. This should not really be surprising, since the bounded and unbounded forms of a queue demand substantially different implementation strategies, particularly in the case of the "static" form based on static data structures. If such an implementation had been adopted for the bounded queue, there would have been little point in inheriting the implementation of *SIMPLE_QUEUE* (4.1) since its body would have to be completely redefined. Irrespective of the implementation strategy, however, it remains true that the methods offered in the interface of a bounded queue are *logically* a superset of those offered by the unbounded form (e.g. *SIMPLE_QUEUE*). This suggests that *BOUNDED_SIMPLE_QUEUE* should indeed be an heir of *SIMPLE_QUEUE* despite the fact that the inherited code may have to be completely redefined.

This apparent contradiction is a symptom of the fact that in DRAGOON, as in many other other object-oriented languages, the inheritance hierarchy is used both to represent logical subclass relationships and practical "implementation" relationships for reusing code. It is not possible to express the fact that two classes logically exist in a subclass/superclass relationship without the heir also inheriting the complete

implementation of the parent. Like other object-oriented languages such as Eiffel, however, DRAGOON provides a facility for overcoming this particular difficulty with the various forms of queues.

The basic problem is that the bounded and unbounded forms of a queue are not genuinely in a subclass/superclass relationship, but actually represent alternative implementation strategies of a single, higher-level abstraction - a queue. To permit this relationships to be expressed more naturally DRAGOON supports the concept of *abstract classes*, which define the interface of a particular abstraction, but leave descendent classes to provide the specific implementations. Just such a class is needed to define the basic abstraction of a queue without carrying any implementation commitments.

```
class QUEUE introduces                                              /4.4/
    function POP return ELEMENT is deferred;
    procedure ADD (ITEM : in ELEMENT) is deferred;
    function IS_EMPTY return BOOLEAN is deferred;
    UNDERFLOW : exception;
end QUEUE;
```

The keyword "deferred" after a method specification indicates that this class does not provide an implementation for the method. This does not mean, however, that abstract classes such as *QUEUE* possess no body. Quite the contrary, even if a class defers all its methods - which is not obligatory - it may still define instance variables and data types in its body. In fact, it may also provide a special kind of "degenerate" body for each of the deferred methods. Such a body does not actually "implement" the method as such, but defines *pre* and *post conditions* that should be satisfied before and after its execution.

4.3.1 Pre/Post Conditions

The pre and post conditions of DRAGOON are similar in style to those of Eiffel - enhanced boolean expressions specifying conditions that should be valid before and after the execution of a method. As in Eiffel, a predefined exception (*CONDITION_ERROR*) is raised if the precondition is not satisfied on invocation of the method, or the post condition is not satisfied on its completion. However, pre/post conditions perform a completely different role in DRAGOON to that in Eiffel, where their main purpose is to define the "contract" between the *client* of a class and its implementor. The idea is that if the client ensures that the pre condition is satisfied before calling a method, its implementor will guarantee that the post condition is satisfied on its completion.

While this view of "programming as contracting" certainly has important benefits for reuse and program correctness in a sequential environment, it runs into difficulties in concurrent systems. This is because the preconditions that need to be satisfied generally depend on the state of the server object, and the client can only obtain the required information by calling a method of the object concerned. A typical precondition of a queue's *POP* operation, for example, is that the queue should not be empty, and since this assertion depends directly on the state of the queue, a client can only ascertain this information by calling the *IS_EMPTY* method beforehand. This presents no problem in a system with only one execution thread, but in a concurrent system, where there may be many concurrently executing clients, there is no guarantee that the state of the object will not alter in the time between the calls to *IS_EMPTY* and *POP*.

Assertions about the state of an object, therefore, do little to support the "programming as contracting" approach in a concurrent environment. Moreover, this approach is not compatible with the DRAGOON philosophy of separating the specification and body parts of a class, since an assertion placed in the specification cannot refer to the state variables, while an assertion placed in the body cannot be seen by a client and serve as the basis of a "contract".

Although assertions are of no value in DRAGOON for specifying a contract between the client and implementor of a class they are useful for specifying a "contract" between the "deferrer" of a method and its implementor in a descendent class. If an abstract class specifies certain pre and post conditions on the body of a method which it defers, then these will apply to all subsequent redefinitions of the method. The designer of the abstract class is therefore conveying information about the intended logical semantics of the deferred methods to future implementors.

As in Eiffel, DRAGOON also supports the definition of "class invariants" in the body of a class, which specify conditions that must be satisfied at all times, particularly after the execution of any of its methods.

4.3.2 Completing Methods

Since by definition an abstract class is not equipped with a full implementation (at least one method must be deferred) it is not possible to generate instances of the class within other DRAGOON objects. This does not rule out the possibility of defining instance variables of an abstract class type, however, since the polymorphism facilities of DRAGOON allow descendent classes to be assigned to them.

Having defined *QUEUE* (4.4) as an abstract class, it is now possible to model the true relationship between the bounded and unbounded forms of the abstraction as separate subclasses providing alternative implementations. However, bounded and

unbounded are not the only forms of queues which need to be represented somewhere in the hierarchy. As mentioned above, bounded forms may be implemented by either dynamic or static data structures, while unbounded queues may either be managed or unmanged. To cater for these additional forms also, it is sensible to introduce two more abstract classes to represent the distinction between the bounded and unbounded forms, but to leave open the other implementation alternatives :-

```
class UNBOUNDED_QUEUE                                          /4.5/
    inherits QUEUE;
end UNBOUNDED_QUEUE;
```

```
class BOUNDED_QUEUE                                            /4.6/
    inherits QUEUE;
    introduces
        function IS_FULL return BOOLEAN is deferred;
        OVERFLOW : exception;
end BOUNDED_QUEUE;
```

Although it may not be immediately apparent from the specification of the first class, *BOUNDED_QUEUE*, both these classes are abstract because they do not provide implementations for the deferred method they inherited from *QUEUE*. *BOUNDED_QUEUE* introduces a new deferred method and so is clearly abstract. It is the descendent classes at the next level in the hierarchy that provide the required implementations, For example :-

```
class UNMANAGED_QUEUE                                          /4.7/
    inherits UNBOUNDED_QUEUE;
    completes
        POP, ADD, IS_EMPTY;
end UNMANAGED_QUEUE;
```

The keyword "completes" is used in the interface to indicate that the class provides complete bodies for the methods listed. In this case, the class implements the unmanaged, unbounded form of the component just like the original class *SIMPLE_QUEUE* above. In fact the body of *SIMPLE_QUEUE* (4.1), would be perfectly satisfactory as the body of *UNMANAGED_QUEUE*. The advantage of using abstract classes in this way is that other implementations can now be defined with the proper relationship to this form, rather than having to inherit from *SIMPLE_QUEUE* and then redefine most of the implementation. For example, a bounded form implemented using dynamic data structures (e.g. *BOUNDED_SIMPLE_QUEUE* (4.3)) is now defined as a subclass of the abstract class *BOUNDED_QUEUE* :-

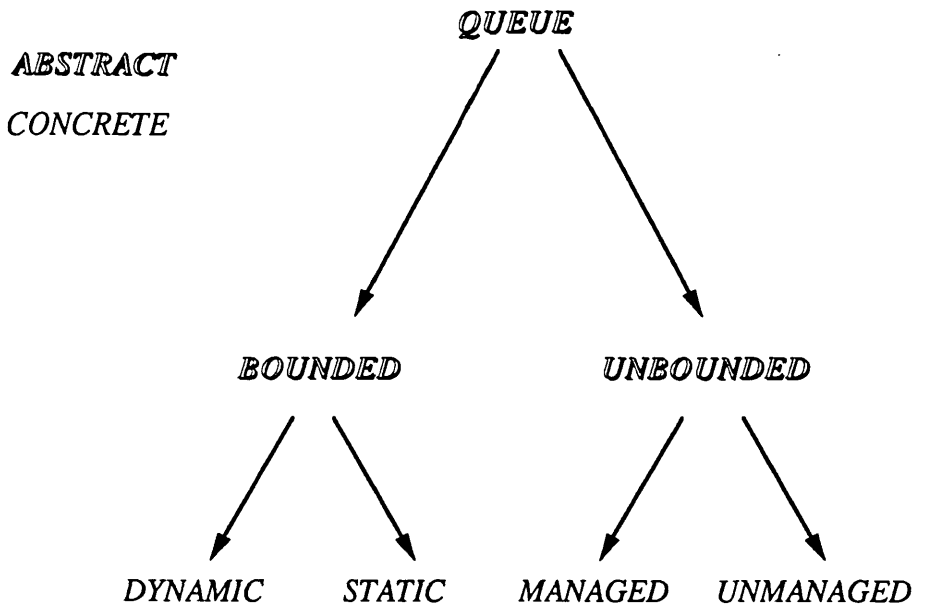


Figure 4.1: Inheritance Hierarchy for the "Queue" forms

```
class DYNAMIC_QUEUE                                          /4.8/
    inherits BOUNDED_QUEUE;
    completes
        POP, ADD, IS_EMPTY;
end DYNAMIC_QUEUE;
```

Similarly, the other two forms - "static" and "managed" - are defined as subclasses of *BOUNDED* and *UNBOUNDED* with specifications identical to *DYNAMIC_QUEUE* and *UNMANAGED_QUEUE* respectively. The final form of the inheritance hierarchy modelling all these different spatial forms of the queue abstraction is thus of the form illustrated in Figure 4.1. Common properties have been factored out and represented by abstract classes.

One of the effects of deferring methods is to decouple the specification of a method from its implementation. In Eiffel, therefore, in which classes must be defined completely in a single module, abstract classes are used as a way of separating the specification part of a class from its implementation. This is not the case in DRAGOON, however, because the specification and implementation of classes are already contained in separate program modules - the specification part and body part. The role of abstract classes in DRAGOON is only to allow the correct logical relationship between classes to be expressed in the inheritance hierarchy, permitting genuinely alternative implementations to be defined as independent peers.

The separation of the implementation and specification parts of a class or method should not be confused with the facility for defining separate subunits either. DRAGOON adopts Ada's approach of permitting the implementation of methods (i.e. subprograms) to be defined in a different program module to the encapsulating unit by replacing the method body with a *stub*. The implementation must eventually be provided in a distinct subunit of the main unit. However, as in Ada, this facility simply serves as a mechanism for simplifying the textual complexity of a program unit, and does not alter the logical structure of the program in any way.

Abstract classes offer one technique for implementing the passive form of an iterator as defined above. The operation which is to be performed successively on each of the elements in the component is simply deferred in the parent class, so that it can be "instantiated" with particular concrete versions in descendents. Although this achieves the basic requirement of parameterising a component class with respect to an operation (the iterator), however, it is not particularly efficient because it requires the explicit reimplementation of the operation for each instantiation. Generic classes, described below, provide a much better way of implementing a passive iterator in DRAGOON.

4.4 Genericity

The discussion so far has focussed on the advantages of inheritance for implementing component families, and has avoided the question of generalising the types of the elements they manipulate by basing the examples on the undefined class type *EXAMPLE*. However, it is clearly unsatisfactory for a strongly typed language such as DRAGOON to have supposedly "reusable" components which are only able to operate on a single type. This would make it necessary to define a new queue component for each different type of queued item, even though the implementation of each component would be identical.

In object-oriented languages it is, in fact, possible to use polymorphism to define a queue component applicable to *any* class type by defining it as a client of *APPLICATION_OBJECT* (i.e. by substituting *APPLICATION_OBJECT* in the above for *EXAMPLE*). Since it is the root of the DRAGOON class hierarchy, every object is a subclass of *APPLICATION_OBJECT*, and can be added to a queue of such objects. This approach, however, completely undermines the typing system of the language, since no constraints are imposed on the types of objects manipulated by the component. While this may be what is required in certain circumstances, in the vast majority of cases it is important to have some checks that the objects manipulated by the component are consistent with its intended usage. As mentioned in the

previous chapter, the resulting reduction in the likelihood of run-time errors and improvements in software reliability is the basic motivation for strong typing.

What is required, therefore, is a mechanism which supports the definition of general components capable of being used with a large number of different types (possibly all), but which also ensures that instances of the component manipulate only certain mutually consistent subsets of these types. In other words, it is desirable to maintain type distinctions when a component is being *used*, but to "factor out" dependencies on particular types in its *definition* [Ichbiah et al, 86].

This facility, sometimes termed parametric polymorphism [Cardelli and Wegner, 85], is supported in its full generality by languages such as ML [Milner, 84]. ML allows functions to be defined with formal arguments of an unspecified type, but when they are invoked checks the mutual consistency of the actual parameters. Typing of this form not only depends on sophisticated inference systems, however, but also makes typing errors less obvious and easy to understand for the programmer. The Ada generic mechanism, on the other hand, provides a static form of parametric polymorphism in which generalised, generic templates have to be explicitly "instantiated" at compile time with the particular type they are to operate on. In one sense, therefore, generics are simply a convenient syntactic device for generating different versions of a component tailored to different types. However, since it enables the compiler to share their implementations, it is nevertheless a highly effective method for increasing the generality of components. Ada allows components to be parameterised not only with respect to types, but also subprograms and values.

As described in chapter 2, object-oriented languages such as DRAGOON already provide a powerful polymorphism mechanism, known as *inclusion* polymorphism, in which instance variables may refer to different class types during the execution of a program. This relies on the fact that a subclass generally inherits all the properties of its parent(s), and can therefore be used in contexts where one of its ancestors is expected.

It is not immediately obvious, therefore, how genericity compares and combines with the inclusion polymorphism available in object-oriented languages, or whether it has anything extra to offer in such languages. Meyer considers this question in detail in his seminal paper [Meyer, 86] and book [Meyer, 89], which explicitly compare genericity as found in Ada with inheritance as found in Eiffel. To answer this question in the context of DRAGOON, the following sections review the different forms of genericity identified by Meyer, and compare these with the alternative generalisation possibilities offered by inclusion polymorphism. After Eiffel's facilities for realising the various forms of parametric polymorphism have been outlined in section 4.4.3, the next few sections describe the approach adopted in DRAGOON.

4.4.1 Constrained and Unconstrained Genericity

As the basis for his comparison, Meyer identifies two main forms of "type" genericity in Ada; constrained and unconstrained.

Unconstrained Genericity. In the unconstrained form the generic component makes no assumption about the type with respect to which it is parameterised. A queue is a good example of such a component since it performs no operations, other than basic assignments, on the elements it stores. It is possible to have queues of any type of element, therefore, although it is likely that particular instances of the queue should only contain elements of a single type. This form of genericity is represented in Ada by parameterising the component with respect to a single *private* data type. For example, an Ada package defining such a queue would have a generic formal part of the form :-

```
generic                                                          /4.9/
    type T is private;
package QUEUE is
    :
```

Specifying the type as private means that the body of the package is only able to perform "basic" operation on it, such as assignments, and membership and equality tests. The component can thus be instantiated with any type providing these facilities (i.e. any data type not defined to be "limited private") :-

```
package INTEGER_QUEUE is new QUEUE (T => INTEGER);               /4.10/
package COLOUR_QUEUE is new QUEUE (T => COLOUR);                 /4.11/
```

These generic instantiations generate new queues specialised to operate on one particular type. Although they are generated from the same template, the strong typing rules of Ada will not allow an inappropriate type to be added to such a queue.

Constrained Genericity. In the case of constrained genericity the component makes some additional assumptions about the operations that can be performed on the type. In Ada, these "constraints" on the type are specified as additional subprograms in the generic formal part of the component. These indicate that the component can only be instantiated with a type for which operations of the required form have been defined. When the instantiation is performed, the associated subprograms must be provided as actual parameters along with the type they operate upon.

Suppose, for example, that a variant of the queue component was required to store "dated items" of the kind introduced in the last chapter (section 3.4.2) in chronological order, so that the item "popped" off such a queue would be the one

with the "lowest" date. Only if several items have the same "date" are they arranged in FIFO order. As mentioned in section 3.4.2, a dated item is any abstraction which possesses at least one *SIMPLE_CALENDAR.DATE* attribute, and associated accessor and update operations, but will usually, of course, possess many more attributes and operations. Since Ada does not support classes, such an abstraction would have to be represented as a conventional abstract data type of the following form :-

```
with SIMPLE_CALENDAR;                                                    /4.12/
package DATED is
    type ITEM is private;
    function DATE (I : in ITEM) return SIMPLE_CALENDAR.DATE;
    procedure SET_DATE_TO (I : in ITEM; DATE : in SIMPLE_CALENDAR.DATE);
        :
            -- other subprograms
private
        :
            -- type ITEM defined as a record with at least
            -- one field of type SIMPLE_CALENDAR.DATE
        :
end;
```

Since the *SIMPLE_CALENDAR.DATE* field is only one of many possible fields the (private) type *DATED.ITEM* could possess there is a potentially infinite number of "dated item" abstractions that could be modelled by a package of this basic form. The special "chronological queue" component mentioned above can be made generic to all these different abstractions, and only these, by a generic formal part of the form :-

```
with SIMPLE_CALENDAR;                                                    /4.13/
generic
    type T is private;
    with function F (I : in T) return SIMPLE_CALENDAR.DATE;
    with procedure P (I : in T; DATE : in SIMPLE_CALENDAR.DATE);
package CHRONO_QUEUE is
    :
```

The formal part of this package indicates that it expects to be provided with a type, and associated subprograms, with the specified signature. In contrast with the unconstrained case, therefore, in which the body of the generic package could only perform the basic (private type) operations on *T*, in this case the body is able to use the additional operations *F* and *P*. Hence this component is "constrained" to being instantiated only with types possessing the required kind of associated operations, such as *DATED.ITEM* :-

```
with DATED;                                                    /4.14/
package DATED_QUEUE is new CHRONO_QUEUE
        (T => DATED.ITEM; F => DATED.DATE; P => DATED.SET_DATE_TO);
```

The essential difference between these two forms of genericity is that in the unconstrained form no special assumptions are made about the generic formal type, while in the constrained form the component is parameterised with respect to a type intimately associated with certain kinds of operations - in other words, an abstract data type. Constrained genericity, therefore, can be seen as a way of supporting parametric polymorphism with respect to *abstract data types.* Unconstrained genericity is merely a limiting case in which the type has no special associated operations.

4.4.2 Inclusion versus Parametric Polymorphism

As described in chapter 3, object-oriented languages provide classes as an alternative means of representing data abstractions such as "dated item" above[3]. In terms of the class-based representation of data abstractions, therefore, genericity can be regarded as providing parametric polymorphism with respect to classes.

Classes, however, are also the source of inclusion polymorphism of object-oriented language. Making a component generic with respect to a class, therefore, combines the mechanisms of parametric and universal polymorphism (the two *universal* forms identified by Cardelli and Wegner [85]) in the same component. Is this desirable, or indeed even safe, however ? To answer this question, consider the alternative ways in which the "chronological queue" component above could be generalised with respect to the "dated item" abstraction using these two forms of polymorphism.

Inclusion polymorphism is achieved simply by making the component (i.e. the class) a direct *client* of the class *DATED_ITEM* (3.10).

```
with DATED_ITEM;                                               /4.15/
class INC_CHRONO_QUEUE is
    function POP return DATED_ITEM;
    procedure ADD (ITEM : in DATED_ITEM);
end INC_CHRONO_QUEUE;
```

Due to the typing rules, any instance of *INC_CHRONO_QUEUE* can manipulate instances of any subclass of *DATED_ITEM* (3.10), such as *DATED_DESCRIPTOR* (3.13) and *DATED_LIBRARY_BOOK* (3.14). *INC_CHRONO_QUEUE* is therefore generalised with respect to the "data abstraction", *DATED_ITEM*, as required.

[3]DRAGOON, of course, permits data abstractions to be represented as classes or abstract data types.

To achieve parametric polymorphism, on the other hand, the class modelling the "chronological queue" component must have the "dated item" abstraction as a formal generic parameter :-

```
generic                                                          /4.16/
    DATED_ITEM;
class PAR_CHRONO_QUEUE is
    function POP return DATED_ITEM;
    procedure ADD (ITEM : in DATED_ITEM);
end PAR_CHRONO_QUEUE;
```

Once again the queue component, *PAR_CHRONO_QUEUE* is generalised with respect to *DATED_ITEM*, but this time it is necessary to instantiate the generic component with the desired version of *DATED_ITEM* first. The result is a class specially tailored to a particular version of the abstraction. Functioning components (i.e. objects) are generated by further instantiating the resulting classes in the normal way.

There are, therefore , two "instantiation" steps involved in the parametric form; the instantiation of the generic component class with the class(es) being manipulated, and instantiation of the resulting class to produce instances. To avoid potential confusion between the two operations the term *actualisation* will be used to describe the "instantiation" of a generic, leaving the term "instantiation" to have its usual meaning with respect to classes. To generate a working chronological queue using parametric polymorphism, therefore, the class *PAR_CHRONO_QUEUE* must first be actualised with a particular version of *DATED_ITEM* (3.10), and the resulting class instantiated.

Since constrained genericity, at least as provided by Ada, only requires that the signatures of the formal and actual operations match, there is no reason why the class provided as the actual parameters of an actualisation of *PAR_CHRONO_QUEUE* should be a subclass of the formal parameter *DATED_ITEM*. *PERISHABLE_DESCRIPTOR* (3.6), for example, possesses precisely the kind of methods expected by the component *PAR_CHRONO_QUEUE*, and therefore could be supplied as an actual parameter of an actualisation even though it is not a subclass of *DATED_ITEM*. However, for reasons explained below, DRAGOON restricts the actual parameter to be a subclass of the formal.

The main difference between the two approaches is that in the first *any* subclass of *DATED_ITEM* can be manipulated by *any* instance of the component *INC_CHRONO_QUEUE*. With the second, however, depending on the particular generic actualisation from which they are generated, different instances of the component can be constrained to manipulate one particular version of the abstraction *DATED_ITEM*.

Consider the following actualisations of *PAR_CHRONO_QUEUE*, for example :-

```
class LIBRARY_CHRONO_QUEUE is new PAR_CHRONO_QUEUE                    /4.17/
                                  (DATED_ITEM => DATED_LIBRARY_BOOK);
class DESCRIPTOR_CHRONO_QUEUE is new PAR_CHRONO_QUEUE                 /4.18/
                                  (DATED_ITEM => DATED_DESCRIPTOR);
```

LIBRARY_CHRONO_QUEUE and *DESCRIPTOR_CHRONO_QUEUE* are actualisations of the basic *PAR_CHRONO_QUEUE* component tailored to different version (i.e. subclasses) of the *DATED_ITEM* abstraction. Consequently, it is not possible to "*ADD*" a *DATED_LIBRARY_BOOK* (3.14) object to an instance of the former, and a *DATED_DESCRIPTOR* (3.13) object to an instance of the latter, despite the fact that they are both subclasses of *DATED_ITEM*. It is possible, however, to "*ADD*" descendents of *DATED_LIBRARY_BOOK* and *DATED_DESCRIPTOR* to the respective *PAR_CHRONO_QUEUE* actualisations.

With inclusion polymorphism, any instance of *INC_CHRONO_QUEUE* can be treated as a queue of *DATED_LIBRARY_BOOK* objects or *DATED_DESCRIPTOR* objects, since both these classes are descendents of *DATED_ITEM*. Precisely because of this, however, there is nothing to stop a *DATED_LIBRARY_BOOK* object being added to a queue supposed to contain only *DATA_DESCRIPTOR* objects, and vice versa. Although this may in fact be what is required in a small number of applications, in the vast majority of cases this is likely to be a logical error.

Parametric polymorphism allows the logical distinction between the different types of queue elements to be made explicit and enforced by strong typing rules, while at the same time allowing their common structure to be modelled in a single component. In such circumstances, therefore, it provides the same advantages of strongly typed languages over untyped ones.

Inclusion polymorphism and parametric polymorphism in the form of genericity are therefore distinct, but complementary, mechanisms providing different facilities to the programmer. However, Meyer [89] describes how something of the effect of genericity can be realised using the inheritance mechanism of Eiffel. This is achieved by making the different specialisations of a component, tailored to different types, subclasses of the base form and redefining their methods to manipulate the required type. In the case of the chronological queue for example, the type-specific versions of the component, *LIBRARY_CHRONO_QUEUE* (4.17) and *DESCRIPTOR_CHRONO_QUEUE* (4.18), would be subclasses of the general form *INC_CHRONO_QUEUE*, each completing the deferred methods *ADD* and *POP* to manipulate the appropriate type (*DATED_LIBRARY_BOOK* and *DATED_DESCRIPTOR* respectively).

This technique, however, depends crucially on Eiffel's facility for redefining a method's argument types, and is not supported in DRAGOON or most other

statically-typed object-oriented languages. Moreover, to ensure complete type consistency a considerable number of the component's methods often need to be redefined for each type specific specialisation. This largely defeats the point of introducing genericity in the first place, which is precisely to avoid the large scale redefinition of components in order to generate copies that are identical except for the type they are tailored to.

Inheritance alone, therefore, does not provide a satisfactory way of achieving parametric polymorphism in an object-oriented language. On the hand, as pointed out by Meyer, genericity provides an even less satisfactory technique for achieving inclusion polymorphism. In a language intended to promote reuse, therefore, it is desirable to provide mechanisms supporting both inclusion and parametric polymorphism.

4.4.3 Declaration by Association

Eiffel addresses this requirement by two distinct techniques. Support is provided for unconstrained genericity by introducing the concept of generic classes. These are parameterised with respect to an "unconstrained" class type, so that the implementor of the component is unable to make any assumptions about the methods of the formal parameter. Constrained polymorphism, on the other hand, is handled by a completely different mechanism known as "declaration by association", allied with "type redefinition", which permits the types of instance variables and formal method arguments to be redefined. An Eiffel-like class using these facilities to model the "generic" class *PAR_CHRONO_QUEUE* (4.16) would be of general form:-

```
class EIFFEL_CHRONO_QUEUE export                                    /4.19/
    ADD, POP;                    -- indicates which methods are exported by the class
feature
    ANCHOR : DATED_ITEM;                                            -- the anchor
    DATA_STORE : array 1 .. X of (like ANCHOR);
        :                                  -- other variables storing aspect of state
    function POP return (like ANCHOR) is
    begin
        :
    end;
    procedure ADD (ITEM : in (like ANCHOR)) is
    begin
        :
    end;
end EIFFEL_CHRONO_QUEUE;
```

As far as possible this class is written in DRAGOON syntax, but employs the Eiffel syntax where necessary. Unlike DRAGOON classes, those in Eiffel do not have a separate specification and body, which means that their methods are defined in the scope of their state variables, such as *DATA_STORE*. This is an important prerequisite for declaration by association, since it allows variables and method parameters to be declared as "like" another variable. In the above class, for example, the elements of the array, *DATA_STORE*, and formal parameters of the "*ADD*" and "*PUT*" methods, are defined to be "like" (i.e. of the same type as) the variable *ANCHOR*, which is provisionally defined to be of class type *DATED_ITEM* (3.10). This variable is called the "anchor" for the associated declarations, and may be used by the classes method's just like any other object of class *DATED_ITEM*. The same is true of the objects declared to be "like" it. In fact, in this particular class, the variable, *ANCHOR*, plays no role other than as an "anchor" for the associated declarations. It is the array object *DATA_STORE* which actually stores the queue.

EIFFEL_CHRONO_QUEUE is a normal class which behaves just like *INC_CHRONO-_QUEUE* (4.15). Instances of the class may store objects of class *DATED_ITEM* (3.10) and any of its subclasses. The difference arises when subclasses of *EIFFEL_CHRONO_QUEUE* are defined. Eiffel permits an heir class to redefine the type of inherited variables to be of any type-compatible class. For example, the variable *ANCHOR* can be redefined to be of class type *DATED_LIBRARY_BOOK* in a subclass of *EIFFEL_CHRONO_QUEUE*, since this is a subclass of *DATED_ITEM* (3.10) :-

```
class EIFFEL_BOOK_QUEUE export                                    /4.20/
    ADD, POP;
inherits
    EIFFEL_CHRONO_QUEUE;
redefines
    ANCHOR;
feature
    ANCHOR : DATED_LIBRARY_BOOK;          -- redefined to a subclass
end EIFFEL_BOOK_QUEUE;
```

Thanks to the "declaration by association" mechanism, however, this not only has the effect of changing the class type of the variable *ANCHOR*, but also of all the other variables and parameters declared to be "like" *ANCHOR*. This, in turn, has the effect of altering the signature of the inherited methods, since the formal parameters were declared by association to *ANCHOR*. The class *EIFFEL_BOOK_QUEUE*, therefore, is a version of its parent specialised to handle objects of class *DATED_LIBRARY_BOOK* (3.14), and thus corresponds to the generic actualisation, *LIBRARY_CHRONO_QUEUE* (4.17), of *PAR_CHRONO_QUEUE* (4.18).

One of the advantages of this approach is that different actualisations of a particular generalised component are handled as subclasses in the inheritance hierarchy. No extra typing rules are therefore required to describe the type-compatibility of such classes. However, this is gained at the expense of employing two completely different mechanisms for what are essentially two different aspects of the same property - parametric polymorphism. Unconstrained genericity is supported by generic classes, while constrained genericity is handled by a combination of "declaration by association" and "type redefinition". Unfortunately this means that the features for supporting constrained genericity in Eiffel "inherit" the typing inconsistencies associated with the "type redefinition" mechanism discussed in the previous chapter [Cook, 89].

4.4.4 Parameterisation Contract

Because of these disadvantages with the "declaration by association" approach DRAGOON adopts a uniform generic-based mechanism in the style of Ada to support parametric polymorphism, as described below. As in Eiffel, therefore, DRAGOON introduces the notion of *generic classes* which are parameterised with respect to one or more types, and must be *actualised* with an appropriate actual type to give rise to a normal class. The important difference between DRAGOON generic classes and those found in Eiffel is the possible forms of their generic formal parts.

As pointed out above, the constraints specified in the formal part of a generic components define a contract between the component implementor, and the component user who "actualises" it for a specific purpose. If the actual generic parameters with which the component is actualised satisfy the contract by providing the services expected in the implementation, the component will be "guaranteed" to function correctly.

Constrained and unconstrained genericity as defined above are in fact just two examples of a range of different "contracts" that can be expressed between the implementor of a generic component and its user. Quite apart from constrained genericity, in which subprograms are specified explicitly as part of the contract, Ada provides direct syntactic support for quite a number of other constraints on generic parameter types [DoD, 83], such as :-

type *T* is private – specifies that only basic operations may be used by the component implementor, such as assignment, equality and inequality comparisons.

type *T* is limited private – more strict than the above in that it forbids also the use of assignment and comparison operations by the implementor. Since these are available with every declared type, including classes, any DRAGOON type can be supplied as an actual parameter.

type *T* is (<>) – in this context the box "<>" denotes a discrete type. Unlike the above, therefore, this constrains the set of types which can be used as actual parameters to discrete (data) types.

type *T* is range (<>) – more strict than the above in that it denotes an *INTEGER* range.

type *T* is delta (<>) – denotes a fixed point type

type *T* is digits (<>) – similarly, a floating point range.

As DRAGOON incorporates the full typing scheme of Ada, it naturally permits all these syntactic constructs to be used as the formal parts of generic classes. This permits a full range of constraints to be placed on the data types used as generic parameters. DRAGOON supplements the range of constraints possible in "constrained genericity" above by permitting classes to be named as generic formal parameters. In DRAGOON, for example, the precise form of the class *PAR_CHRONO_QUEUE* (4.16) discussed above would be :-

```
generic                                                          /4.21/
    type T is DATED_ITEM;
class PAR_CHRONO_QUEUE is
    function POP return T;
    procedure ADD (ITEM : in T);
end PAR_CHRONO_QUEUE;
```

with particular "actualisations" such as *LIBRARY_CHRONO_QUEUE* and *DESCRIPTOR_CHRONO_QUEUE* having a similar form as in (4.17) and (4.18) above, but with "*DATED_ITEM*" replaced by "*T*".

The important difference between the generic classes in DRAGOON and those in Eiffel is that in the latter the generic parameters of a class are essentially "private" in the Ada sense. Although class types may be supplied as actual parameters the generic class is not able to make use of this fact in its implementation. In DRAGOON, however, a class which is generic with respect to another class type is permitted to call all the methods of the formal class type, because it can only be actualised with classes which are type-compatible (i.e. subclasses) of the formal parameter. Generic classes of the kind supported in Eiffel, which can be actualised with a class or data type, correspond to DRAGOON classes having private formal parameters. Classes which are generic with respect to an anonymous class type (i.e. which must be actualised with a class type, but do not use its methods) can be defined in DRAGOON as classes with a formal parameter of class *APPLICATION_OBJECT*. This class possesses no methods (other than the special *CREATE* operation) and so when used as a generic formal parameter in a sense represents a "private" class.

Since this is also the ancestor of all (non-generic) classes in the library, however, it can be matched by any of them.

Section 4.3 demonstrated how a passive iterator could be implemented (albeit in a not particularly satisfactory fashion) in DRAGOON using abstract classes. The introduction of generic classes permits a much more efficient implementation, however. In Ada a passive iterator is implemented as a generic package having the iterator operation as a generic parameter. The corresponding effect is achieved in DRAGOON by means of a class that has as a formal parameter an abstract class exporting the iterator operation as its only method. The generic class is free to use this method in its implementation, and can be provided with different versions of the iterator operation by being actualised with different concrete subclasses of the formal parameter.

This is one example of the way in which Ada's facility for parameterising components with respect to subprograms can be simulated in DRAGOON. The final kind of generic parametrisation supported in Ada is with respect to actual data values rather than types, and is used in order to provide initial values and constants such as those needed to model an algebra [Meyer, 88]. This effect can already be achieved in DRAGOON, however, by "redefining" the *CREATE* operation.

4.4.5 Inheriting Genericity

Using genericity to support all forms of parametric polymorphism as described above has the advantages of simplicity and uniformity. There is, however, one major difference between DRAGOON's support for "constrained" parameterisation by means of generic formal class types, and Eiffel's approach using declaration by association. Parameterised classes in Eiffel are amenable to incremental development by means of the inheritance mechanism, but this is not the case with DRAGOON generic classes discussed so far.

There is no fundamental reason, however, why DRAGOON generic classes should not be enriched by the mechanisms of inheritance just like any other class type. Adding new methods to the interface of a generic class by inheritance does not alter the fact that certain parameters of the class are formal, and therefore must be associated with an actual parameter before an executable version is generated. There should be no problem in defining heirs to generic classes, provided these are also regarded as generic.

Suppose for example, that a new version of the generic *PAR_CHRONO_QUEUE* (4.21) component were required with an additional method, *NUMBER_WITH_DATE*, returning the number of stored objects with a given date. This can be generated in DRAGOON by defining an heir of *PAR_CHRONO_QUEUE* which extends its interface

with the required method.

```
class EXTENDED_CHRONO_QUEUE                                        /4.22/
    inherits PAR_CHRONO_QUEUE;
    introduces
        function NUMBER_WITH_DATE (D : in SIMPLE_CALENDAR.DATE)
                                                    return NATURAL;
end EXTENDED_CHRONO_QUEUE;
```

Although this class has no explicit generic formal part like other generic classes, it is nevertheless generic by virtue of the fact that its parent is generic. In other words, the class has inherited the generic formal part of its parent, and therefore needs to be actualised with precisely the same number and type of actual parameters. Not all the generic parameters have to be inherited, however. A generic subclass may have its own generic formal part which extends the set of generic parameters. Multiple inheritance from generic classes it not permitted.

4.4.6 Type Compatibility of Generics

The remaining question to answer in connection with generics is how they fit in to the typing scheme of the language. Firstly, it is important to note that generic classes themselves are not "first class" types of the language; they cannot be directly instantiated, they cannot act as method parameters, and it is not possible to declare an instance variable of a generic class. Actualisations of generic classes, however, are normal classes for which all these operations are possible.

There are essentially two approaches for defining the type compatibility of generic actualisations. In one approach, which is adopted by Eiffel, generic actualisations are regarded as type-compatible if they conform. Formulating rules to ensure that this is so, however, is even more complicated than in the non-generic case, since it not only depends on the reverse conformance of generic parameters defining method argument types (i.e. "in" parameters), and the forward conformance of generic parameters defining result parameter types and instance variables, but also requires that no parameter is used for both [Cook, 89].

Once again, however, it is questionable whether the flexibility provided by such a definition of type compatibility justifies its complexity. This is particularly so in the case of generic classes, where the reason for choosing genericity (i.e. parametric polymorphism) rather than the usual inclusion polymorphism for generalising a component is to distinguish statically between the types of elements assigned to them. This advantage is largely lost, however, if the different actualisations of a generic component are regarded as type compatible, because there is no way of

ensuring that an instance of one actualisation will not be assigned to an instance variable on which operations only suitable for instances of the other are performed.

Once again, therefore, DRAGOON typing rules for generic actualisations are much more straightforward than those of Eiffel. In DRAGOON a class which is generated by actualisation is *not* type-compatible with any other class[4]

4.4.7 Method Overloading

The two forms of polymorphism discussed above, inclusion and parametric, are termed the universal forms of polymorphism by Cardelli and Wegner [85], since they are manifestations of natural polymorphism. DRAGOON also supports the other category, ad hoc polymorphism, in the form of method overloading. Like subprograms in Ada, methods may be given the same name provided they are distinguishable by the types of their formal parameters.

A good example of where this facility comes in useful is provided by the data base object in the supermarket control system. The job of this object is simply to store *PRODUCT_DESCRIPTOR* objects (3.4) generated by the central controller, and to enable the checkout points, as well as the central controller, to retrieve product descriptors by means of their attributes. The most common attribute used for retrieving product descriptors is their bar code, since this is the only attribute guaranteed to be unique, and which is recognised by the laser reading facilities of the checkout points. However, there will be occasions when it is desirable to retrieve product descriptors by other attributes, such as the label. To cater for these different retrieval possibilities it is necessary to provide a different method for each different attribute key. However, since all the attributes are of different types (partly because of the use of derived *INTEGER* types), it is not necessary to provide each method with a different name. The class modelling the data base, therefore, may have an interface of the form :-

```
with PRODUCT_DESCRIPTOR, PRODUCT;                                    /4.23/
class DATA_BASE introduces
    procedure ADD (P : in PRODUCT_DESCRIPTOR);
    function RETRIEVE (PRICE : in PRODUCT.VALUE)
                                              return PRODUCT_DESCRIPTOR;
    function RETRIEVE (BAR_CODE : in PRODUCT.CODE_NUMBER)
                                              return PRODUCT_DESCRIPTOR;
    function RETRIEVE (STOCK_LEVEL : in PRODUCT.STOCK_NUMBER)
                                              return PRODUCT_DESCRIPTOR;
    function RETRIEVE (LABEL : in PRODUCT.LABEL)
```

[4]It is, of course, type-compatible with itself.

```
                                    return PRODUCT_DESCRIPTOR;
    end DATA_BASE;
```

The *ADD* method is used by the central controller to add new *PRODUCT_DESCRIPTOR* (3.4) objects to the data base, while the purpose of the *RETRIEVE* methods is to return a product descriptor with an attribute of the specified value. It is not specified which object is returned if there is more than one with the appropriate attributes, but if there are none, the null reference is returned.

4.5 Dynamic Binding

The first few sections in this chapter have described how inheritance and abstract classes can be used to minimise the quantity of replication involved in the generation of families of reusable components, while the previous section has described how genericity can be used to increase their generality. All these features are concerned with the design and *construction* of reusable classes - that is, with design *for* reuse. Object-oriented programming languages, however, also have some very powerful facilities for supporting programming *with* reuse.

In the "component engineering" approach to reuse considered so far, the mechanism by which components are reused is *clientship.* If a class in a new system wishes to employ the services of a reusable component, such as the *QUEUE* (4.4) component developed in this chapter, it defines an instance variable of the component class and generates the required instance. The services of the component are therefore exploited by method calls.

As pointed out in the previous section, one of the key ideas of the object-oriented approach is to decouple the act of invoking a method from that of executing it. When combined with the polymorphism properties of object-oriented languages, and the facility for redefining methods in descendent classes, this separation of concerns gives rise to another extremely powerful feature for reuse known as *dynamic binding.*

Consider, for example, an object which needs to use a queue component in its implementation. As described above, there is a whole family of different variants and forms of the basic queue abstraction, of which those discussed above, and illustrated in 4.1, are but a small number. With a conventional Ada implementation of the component family, the designer of the client software must decide upon one particular implementing package to use, and must explicitly "with" that package in the client code. This is fine if the context does in fact require one particular variant or form of the component, and the designer is able to identify this form at the coding stage, but in many cases this is not the case. The client software is often not concerned

which particular form of queue is used, and even if it is, the designer may not be aware of precisely which form is required at the coding stage.

The dynamic binding properties of object-oriented languages such as DRAGOON overcome this problem, because the client software can be completely decoupled from any consideration about the implementation of the reused component. Although a client object may actually invoke the services of a queue, for example, through an instance variable of class type *BOUNDED_QUEUE* (4.6), due to the assignment rules of the language any subclasses of *BOUNDED_QUEUE* may actually service the method calls at run-time. The form of the queue component to which the instance variable refers, and hence the specific way in which the methods accessible through the variable are implemented, is determined dynamically by the previous assignments. In other words, the particular version of a method that services a method call is dynamically "bound" to the method name.

This feature is particularly powerful in connection with abstract classes, since as described above, they enable common, implementation-independent features of classes to be factored out and modelled at a higher level of abstraction. Thus if a client object were not concerned which particular form or variant of queue it used, but simply wished to use a queue of any kind, it would be a client of the abstract class *QUEUE* (4.4) above. Although this is an abstract class, it is possible for the client software to define an instance variable of the class and invoke its methods, because at run-time the instance variables will refer to instances of its concrete subclasses. The class *QUEUE*, therefore, would be used by client software with no particular implementation requirements, while clients with more specific implementation needs would use other classes at a lower level of abstraction. For example, a class requiring a bounded queue would be a client of *BOUNDED_QUEUE* (4.6).

Augmented by abstract classes, therefore, dynamic binding permits client software to use components of precisely the desired level of abstraction, and be completely unaware of the particular object which actually services the method class. Moreover, as new implementations of the component are added to the library, the type of the object implementing the methods can be changed without affecting the client. This contrasts sharply with the situation in Ada, in which it is not possible to change the form of component used without modifying the client code[5] because it must name the implementing package explicitly.

[5] Ada is better than most non-modular languages in this respect, because it at least permits the redefinition of the package body.

4.5.1 Extensibility

As pointed out by Cox [86] and Meyer [88], and demonstrated by the above example, the main advantage of this feature from the reuse point of view is the extensibility of software systems - the ease with which they can be adapted for new application. In conventional procedure oriented languages such as Pascal and Ada, implementation dependent aspects of design are dispersed widely amongst components of a system, even when an object-oriented design method is adopted. This means that the consequences of a change or extension to one part of the system have an impact throughout. In contrast, the uncoupling of method invocation from execution in object-oriented languages, and polymorphism permitted by the typing scheme, localises implementation details, and stops them spreading beyond those components that actually need to know them.

Consider, for example, a client of the class *PRODUCT_DESCRIPTOR* (3.4) defined in the previous chapter. This may declare an instance variable of the class :-

```
    :
P : PRODUCT_DESCRIPTOR;
D : SIMPLE_CALENDAR.DATE;
    :
```

and may invoke any of the methods exported in its interface :-

```
    :
D := P.SELL_BY_DATE;
    :
```

Instances of any subclass of *PRODUCT_DESCRIPTOR*, such as *ELECTRICAL_DESCRIPTOR* (3.8), *PERISHABLE_DESCRIPTOR* (3.6) and *REDEFINED_DESCRIPTOR* (3.9) may be assigned to the instance variable *P*. Moreover, any method calls expressed using *P* are unaffected by such an assignment, because the body which will service any request is dynamically bound to the call. Since the class *REDEFINED_DESCRIPTOR* redefines the method *SELL_BY_DATE*, the version of the method executed by the above method call depends dynamically on the type of the object to which the instance variables refers.

This contrasts sharply with the way in which the corresponding situation would have to be programmed in Ada. It is possible to achieve a certain degree of polymorphism using variant records, the different sets of attributes of the various kinds of descriptors being represented as different variants of a single record type :-

```
    :
type DESCRIPTOR_OBJECT;
    :
```

```
type DESCRIPTOR_STATE (FORM : PRODUCT_TYPE) is record                /4.24/
                                          -- type "PRODUCT_TYPE defined in (3.1)
    BAR_CODE : PRODUCT.CODE_NUMBER;
    PRICE : PRODUCT.VALUE;
    STOCK_LEVEL : PRODUCT.STOCK_NUMBER;
    LABEL : PRODUCT.LABEL;
    case FORM is
        when PERISHABLE =>
            DATE : SIMPLE_CALENDAR.DATE;
        when ELECTRICAL =>
            LABEL : PRODUCT_DESCRIPTOR;
        when others =>
            null;
    end case ;
end record;
    :
```

However, once the type has been fixed, it is not possible to add a new kind of descriptor to the system without changing this type definition, and all the procedures (methods) which use it, involving at least the recompilation of all the client software.

The situation with respect to dynamic binding is even worse. In order for the client software to dynamically select between one of these implementations depending on the particular descriptor variant assigned to the access variable, it would have to explicitly select the version required using a case statement controlled by an appropriate enumeration type, *VERSION_ENUMERATION*, say :-

```
    :
case VERSION_ENUMERATION is                                          /4.25/
    when BASIC =>
        -- call the original version
    when REDEFINED =>
        -- call the redefined version
    when ... :
end case ;
    :
```

This results in details about the various different implementations being embedded throughout the system in any client software which uses their services. It is not possible to add a new product descriptor to the system, therefore, without modifying large parts of the system.

Clearly, by minimising the extent to which client software is coupled to implementation details of components, object-oriented programming significantly improves the prospects for the incremental modification and extension of software, which is such an important aspect of reuse.

4.6 Programming by Difference

So far the discussion in this chapter has been concerned with various aspects of the basic "component engineering" or "compositional" approach to reuse, in which the process of software development is at least implicitly divided into two aspects: programming *for* reuse, and programming *with* reuse [Sommerville et a et all, 89]. It has been shown how the features of object-oriented languages in general, and of DRAGOON in particular, greatly enhance both these facets of reuse.

Although the "component engineering" approach is an important tool in the quest for reuse, however, it is not by any means a panacea. In spite of the significant advantages of inheritance for developing component families, there is still a practical limit to the number of different variants and forms of components that can be defined and stored in a library, and the degree to which these can be generalised. Situations are bound to arise, therefore, in which the basic abstraction represented by a family of components would be useful in a particular application, but none of the existing components provides precisely the required mix of properties, or is parameterised in the precisely the right way to be used. It is impossible for a component designer to predict and cater for *all* the possible contexts in which the abstraction may be potentially applicable. This is more so for larger more complex objects with a large number of potential variants, and applications.

The component engineering philosophy works well, therefore, for simple, well understood program abstractions, but it is difficult to generalise larger components with highly complex functionality without making them extremely difficult and unattractive to use. The inheritance mechanism of object-oriented languages comes to the rescue again however by also supporting a fundamentally different approach to reuse commonly termed "programming by difference". Although, in the "component engineering" approach described above, inheritance plays an important role in *producing* components by reducing the replication of code involved in component families, clientship is the mechanism which is employed to actually use the components. New objects are thus constructed by *combining* existing components in different ways. In the alternative approach, however, inheritance is employed as the mechanism of reuse, and new objects are generated by adopting or extending old ones.

In the "programming by difference" approach it is not necessary for a component designer to try and predict the various potential uses of a component, and provide it with large numbers of parameters or component forms to make it adaptable for those anticipated, because objects are tailored to new situations "*at the time of reuse*" rather than "*at the time of definition*". If an object is *almost* suitable for a

given application, but does not have exactly the right properties, inheritance allows the programmer to generate a new version of the component with the necessary additions and modifications.

Since any class can be modified or extended by inheritance, and is thus potentially reusable, "programming by difference" blurs the traditional distinction between the process of designing components - programming for reuse - and the process of resuing them in new systems - programming with reuse. This feature is termed the "open and closed" principle by Meyer [89], and describes the fact that a class represents both a concrete ("closed") component that can be directly instantiated for use in a system, and at the same time a modifiable ("open") component that can be adapted for reuse in different contexts. In fact, "programming by difference" is a technique which has been applied on an informal basis by individual programmers for many years. Inheritance is essentially nothing more than a formal linguistic mechanism supporting an informal mode of reuse which is probably that most commonly performed - the editing of existing code to adjust it for a new application.

Chapter 5

Concurrency

Although the "sequential" features of DRAGOON outlined in the last two chapters are combined and presented in a novel way, it is in its support for concurrency and distribution that DRAGOON differs more radically from other object-oriented languages. In order to provide a framework for a discussion of concurrency in the context of object-oriented programming, this chapter first provides a general overview of the main techniques employed in conventional languages. It then goes on to review how these techniques have been employed in the various other object-oriented languages supporting concurrency, before describing in detail the particular approach adopted by DRAGOON. The issue of distribution is addressed in the following chapter.

5.1 Approaches to Concurrency

In concurrent systems computations are performed by the co-operation of several independent (asynchronous) *execution threads*, assumed to have indeterminate (or unrelated) execution speeds. It is this indeterminacy which provides the opportunities for parallel execution in multi-processor systems, with all the attendant performance advantages, but which also demands well-defined synchronisation and communication protocols to ensure safe interaction.

In principle, it is possible for intelligent compilers to spot potential parallelism in normal (i.e. sequential) programs, and to generate concurrent execution threads transparently, but in practice this has proved feasible only for certain limited kinds of numerical applications. For more general applications, where it is necessary to model coexisting entities in the problem domain, the programmer must be aware of the scope for concurrency in a given context, and design the software accordingly. Languages intended to support the explicit description of concurrency in such applications must have mechanisms enabling the specification of :-

- concurrent execution threads,
- inter-thread communication,
- inter-thread synchronisation.

Concurrent Execution Threads

The imperative (Von-Neumann) model of software execution is inherently sequential - a computation being performed as a linear sequence of atomic machine instructions. Consequently imperative languages wishing to support explicit concurrency must allow the programmer to identify program segments which may execute in parallel. This can either be achieved by mechanisms for the splitting, and subsequenting joining, of otherwise sequential execution threads, such as cobegin/coend [Dijkstra, 68] and fork/join [Conway, 63], or by identifying sequences of statements, often termed *sequential processes* or *tasks*, which will execute in parallel with each other. Some languages permit only a fixed number of such sequential processes to be activated at the start of a program (static process model), while others permit the dynamic generation of new processes at arbitrary points (dynamic process model).

Communication

Mechanisms for supporting communication between separate execution threads have traditionally been placed into two categories: those based on the use of shared memory, and those based on the concept of message passing. In the first approach, execution threads communicate by updating shared data structures, and consequently rely on some accompanying synchronisation mechanisms to enforce a sensible access protocol. This approach incurs significant implementation overheads in environments where the communicating processes do not actually have access to shared memory, such as a loosely-coupled distributed system.

In the message passing approach, on the other hand, processes communicate by transferring data values which do not require permanent shared storage. This approach, therefore, is practical in both distributed and non-distributed environments, although it usually implies greater overheads than the first in environments possessing shared memory.

The notion of message passing in concurrent systems should not be confused with the terminology commonly used in object-oriented languages (but avoided in DRAGOON). In the former "message passing" usually implies simply the uni-directional transfer of data, whereas in the latter "message passing" denotes the invocation of a procedure, and thus involves the execution of the servicing code, as well as bi-directional data exchange.

There are many detailed variants of the basic message passing communication scheme, depending on such things as the naming conventions for identifying the communicating parties, and the accompanying synchronisation policies. In a fully *synchronous* message scheme, such as that employed by CSP [Hoare, 78] or Occam [May and Shepherd, 85], to communicate both parties must reach a predetermined synchronisation point before communication may take place - in other words, they must *rendezvous*. The first correspondent to arrive at the point is blocked until the other reaches its corresponding synchronisation point. At the other end of the spectrum are fully *asynchronous* schemes in which either correspondent may perform a "send" or "receive" operation at any time, irrespective of the execution state of the other. In its pure form this approach, therefore, requires unbounded buffer space to store all the messages that could be sent before being processed. In between these two extremes are schemes with limited buffering capacity in which one of the primitives (usually the receiving) is always blocking (i.e. synchronous) while the other (usually the sending) is nonblocking (i.e. asynchronous) so long as there is free space in the buffer.

Synchronisation

Closely related to the communication mechanisms are those for synchronisation. The most fundamental form of synchronisation needed in concurrent systems is *condition synchronisation* where one processes is suspended until some condition is detected or caused by another. With a message passing communication scheme, this can be achieved quite straightforwardly by arranging for the process to be suspended on a receiving primitive, waiting for the arrival of the appropriate message (usually called a *signal* if no data is transferred). Shared variable schemes, on the other hand, have the additional problem of *exclusion synchronisation* in which the shared resources used for communication must be protected from erroneous concurrent access. The basic aim of this form of synchronisation is to define "critical regions" of code which behave like indivisible operations so far as concurrent execution threads are concerned.

It is possible, in the shared variable approach, to realise both forms of synchronisation using complex access protocols [Peterson, 81], but an efficient and natural implementation of exclusion synchronisation ultimately relies on hardware support for indivisible "test-and-set" operations. One of the first linguistic concepts to employ such a facility were *semaphores* introduced by Dijkstra [68], but many alternative models have since emerged, such as conditional critical regions [Brinch Hansen, 73]. A semaphore provides indivisible operations for reading and modifying its value, and an associated queuing mechanism to block processes until a particular value

is reached. In their general form they are actually powerful enough to tackle all synchronisation requirements, but because the processes wishing to access a shared resource are given the responsibility of calling the semaphores in the correct order, they are tricky and cumbersome to use for all but the most trivial problems. Moreover, the erroneous use of a semaphore by just one of the client processes can not only compromise the integrity of the shared resource, but also the freedom from deadlock of the entire system.

To remove the responsibility for handling exclusion synchronisation from the client of a resource, the concept of a *monitor* was introduced [Hoare, 74]. A monitor tightly encapsulates shared date structures so that they can only be accessed by the official procedures guaranteed to provide safe access. Communicating processes, therefore, are no longer concerned with ensuring exclusion synchronisation, but simply request the service they require, and are suspended on a queue of calls until execution of the corresponding operation is permitted.

Using monitors to encapsulate and protect shared resources is a much safer way of achieving exclusion synchronisation than semaphores, but unfortunately is not readily compatible with condition synchronisation [Andrews and Schneider, 83]. The exclusion protocols necessary to ensure safe access to a variable often tend to conflict with those needed to use it as a flag for signalling a condition. Consequently, languages that have adopted the monitor concept as a means of resource protection usually have to incorporate an additional mechanism for handling condition synchronisation.

One approach uses the concept of *condition variables* [Wirth, 82], which have a syntactic similarity to semaphores, but which override the access policy of the monitor when it conflicts with the requirements of condition synchronisation. An alternative solution is to generalise the monitor concept by the introduction of *guards*. These permit the selection of calls to be determined by the state of the monitor, rather than simply by the mutual exclusion criteria. This, in turn reflects the previous access history of the resource.

Guards have found application in languages as disparate as Occam, Ada and PARLOG, but have the drawback that they require an independent execution thread to exist within the "monitor" in order to evaluate them repeatedly. Such monitors, therefore, must be active processes, and hence are often combined with the mechanisms for generating threads of control into a unified process model.

5.2 Concurrency in Object-Oriented Languages

. There are essentially two basic strategies for introducing concurrency features such as those described above into the "sequential" object-oriented programming paradigm described in the last few chapters. One approach is to superimpose the concurrency constructs as an extra layer, *orthogonal* to the object-oriented programming features, while the other approach attempts to fully *integrate* them.

As pointed out by America [89], the first, "orthogonal" approach is usually adopted by languages such as Smalltalk and Trellis/Owl [Schaffert et al., 86] which were originally intended to be sequential. This strategy is certainly effective, but because the concurrency constructs are completely independent of the encapsulation and communication mechanisms of the (normal) objects in a system, it has the major disadvantage that programmers are required to work at two different levels of abstraction [Yokote and Tokoro, 86].

The other "integration" approach is much more in keeping with the general goal of unifying different concerns through the notion of objects, and is adopted by the many object-oriented languages designed for the purpose of supporting concurrency. As might be expected from the large number of traditional concurrency strategies outlined above, however, there are also quite a number of different ways in which this goal can be achieved. These are best considered in relation to the three main criteria outlined in the previous section.

Concurrent Execution Threads

Smalltalk provides the classic example of the "orthogonal" approach, and employs one of the earliest techniques for generating multiple execution threads - forking. An otherwise sequential "block" is split into two threads by sending it a "fork" message, resulting in the generation of a new "process" object.

For languages in the "integration" camp, on the other hand, there are essentially two basic techniques for generating multiple execution threads in a way compatible with the sequential object structure of a system [America, 89].

Asynchronous Method Execution - In the first approach, a new execution thread is generated to execute the body of a method in response to an invocation request (message). The client object, therefore, is free to proceed (asynchronously) with the method execution as soon as the call is accepted. This approach is employed by a number of languages such as Hybrid, but its most notable application is in the Actor languages [Agha, 86].

The actor model of concurrency combines asynchronous method execution with the functional style of generating persistent objects by recursive *behaviour replace-*

ment. Whereas in the sequential object-oriented model the precise functionality of a method changes according to the *state* of the associated object, in the actor model this effect is achieved by changing the "behaviour" (i.e. the way in which messages are processed) inbetween method calls. Messages directed to a particular actor are stored in the order of arrival until they can be processed by one of the actor's behaviours. When the message is processed, this behaviour must specify which of the other behaviours will be used to process the next queued message. Different messages, therefore, are processed by different behaviours of the actor, depending on the previous history of message arrival. As well as specifying its "replacement", a behaviour may generate new actors and send messages to others (including itself).

Active Objects - In the other approach, instead of generating a separate execution thread for each method invocation, a permanent thread is associated with the whole object. In other words, as well as being a unit of modularity and encapsulation, an object is also regarded as an *active* process. This approach, therefore, unifies the notion of "sequential processes" as found in traditional concurrent languages with the concept of "objects" from the object-oriented world.

The difference between these two approaches is essentially one of "granularity". The first approach, characteristic of the Actor languages, offers much "finer grained" concurrency than the second, since it provides parallelism at the level of individual operations, rather than for a set of operations associated with a given object, and the processing of the next queued call may begin as soon as the replacement behaviour has been specified. Actor languages, therefore, are very popular in systems where maximum concurrency is important. The active object model, on the other hand, is more appropriate for embedded systems and simulation purposes, where the objects model or control physical devices with a limited capacity for performing parallel actions [Kaiser et al, 89]. Concurrency in such systems, therefore, is of a larger "granularity" and much more naturally expressed at the level of objects. This approach is adopted by a significant number of languages including Emerald, POOL, Concurrent Smalltalk [Yokote and Tokoro, 86] (an attempt to *integrate* concurrency features with Smalltalk, rather than *superimpose* them), Orient84/K [Ishikawa and Tokoro, 87], ABCL/1 [Yonezawa et al., 86], as well as DRAGOON.

Communication

Languages adopting the "orthogonal" approach such as Smalltalk usually rely on the sharing of objects as the means of supporting communication between separate execution threads. Since this communication technique is commonly employed by normal objects in the system, it does not itself conflict with the object-oriented style. It is the associated synchronisation techniques that are more difficult to integrate

successfully with the object oriented paradigm, as explained below.

In the "asynchronous method execution" technique for generating concurrent threads, because the client object proceeds asynchronously with the execution of methods, it is not possible, in a single transaction, to transfer information from the server to the client on completion of the method body. Such transactions, therefore, reduce to the simple uni-directional message passing model described above - any required data exchange from the server to the client having to be programmed as a separate transaction. Asynchronous message passing of this kind can also be adopted in the "active object" model of concurrency - messages being queued until they can be executed within the thread of control of the *active* receiving object.

"Message passing" in the object-oriented sense actually conforms much more closely to the traditional "remote procedure call" model of distributed systems, involving two way communication and interposed code execution. Most object-oriented languages adopting the "active object" approach to concurrency, therefore, (including DRAGOON) view the method call as a form of "remote procedure call" mechanism. Some, such as POOL, Concurrent Smalltalk and MELD [Kaiser et al, 86] offer both the synchronous remote procedure call and asynchronous message passing forms of communication.

Synchronisation

The most difficult aspect of concurrency to handle satisfactorily in object oriented languages is synchronisation. In those languages following the "orthogonal" philosophy, such as Smalltalk and Trellis/Owl, separate execution threads communicate by means of shared objects. These languages, therefore, have to employ traditional exclusion synchronisation techniques for handling access to the shared resources. Smalltalk, for example, introduces the concept of semaphore objects while Trellis/Owl adopts a variant of "conditional critical regions" known as *lock blocks*. Since there are no constraints on the activation of methods (normal sequential semantics apply), the programmer is responsible for controlling access to critical regions by encoding the appropriate checks in the body of methods. This approach, therefore, suffers from the same serious drawbacks of complexity and safety as the original low-level synchronisation approaches.

In the "integration" approach, communication is by means of method calls (or simpler message passing in the asynchronous case), so the most natural strategy for achieving exclusion synchronisation is to adopt a monitor-like approach, in which method calls (messages) are queued outside the encapsulation boundary of a server object until such time as they may be safely executed. Regarding objects as "monitors" in this way naturally extends their resource encapsulation properties, originally

designed for abstraction and modularity purposes, to handle access control in concurrent environments. This approach, therefore, represents a further unification of the concept of an object with the generalised notion of a monitor. The main characteristic of this approach is that method activation does not take place as soon as an invocation request (message) is received, as in the other approach, but only when the receiver deems that it is safe.

Kafura and Lee [89] identify two main ways of realising this monitor based approach. The majority of languages adopt a "centralised" technique in which the synchronisation conditions determining the permitted method interleavings are specified in a single, "centralised" execution thread, which is usually unified with the process part responsible for making objects active. In POOL, for example, the process part of an active object (called the "body") specifies the conditions under which particular methods may be activated using guarded select statements and special "receive" primitives. In this respect the concurrency model of these languages is very similar to that of Ada - communication taking place between active processes (tasks in Ada) only when both parties are ready and willing to rendezvous. Other languages adopting this centralised approach include ABCL/1[Yonezawa et al, 86], Orient84/K [Ishikawa and Tokoro, 87], ACT++[Kafura, 88] and ALPS [Vishnubhotta, 88].

In the other more *decentralised* approach, which is adopted by the Actor languages and Hybrid [Nierstrasz, 87], the synchronisation constraints determining when queued calls may be accepted and serviced are dispersed amongst the methods rather than concentrated in a single construct. In Hybrid, for example, pending method calls are stored in *delay queues* until the execution of another method opens the queue and allows the call at the head to be executed.

The main differences between these two approaches is that the centralised version requires the monitor objects to be active, since is relies on guard-like constructs, while the decentralised version does not.

5.2.1 Exclusion Synchronisation and Inheritance

Since DRAGOON was designed to conform as closely as possible to Ada it might have been thought that the language would have attempted to integrate Ada's tasking constructs with the object-oriented programming model. For example, tasks might have been used for the active part of objects, entry calls as the means of inter-object communication, and (as in POOL and ALPS) the rendezvous, coupled with guarded select statements, to provide exclusion synchronisation. As pointed out by Wegner [87] and Kafura and Lee [89], however, this approach, like all the others mentioned above, cannot be satisfactorily combined with the inheritance mechanism.

The conflict between exclusion synchronisation and inheritance in these approaches arises because they require the synchronisation conditions to be explicitly encoded in the source code of classes. Whether they are specified centrally within the process part of a class, or dispersed amongst its methods, exclusion constraints, by their very nature, must involve *all* the methods in the interface of an object. When a class is generated by inheritance, however, the inherited code is unware of the existence of any new methods which may have been introduced. The inherited synchronisation conditions, in particular, do not extend to any methods introduced by the heir. Appropriate constraints covering all the methods in the heir's interface can only be produced either by rewriting the centralised control thread (e.g. the body in POOL) or all the inherited methods (e.g. Hybrid). In either case, the quantity of redefinition involved almost entirely defeats the reuse benefits of inheritance.

The "orthogonal" approach of Smalltalk, which uses semaphores to protect critical regions, is actually more compatible with inheritance than the monitor-based approaches described above, because each method is directly responsible for handling the access constraints which apply to it. It is possible, therefore, to include the appropriate operations (e.g. signal and wait) in the new methods introduced by an heir class without having to redefine the inherited methods. However, the already well known difficulty of programming with semaphores is greatly compounded in this approach since the programmer must ensure that semaphore manipulation by the new methods is compatible with that of all the inherited methods.

Because of these difficulties, few languages to date have managed to combine concurrency concerns with a fully general inheritance mechanism. The only languages which do incorporate facilities for both are those such as Smalltalk which have adopted the "orthogonal" approach, and which therefore do not attempt to integrate them. Languages such as Emerald, POOL and ALPS which have concentrated on the supporting concurrency entirely within the object-oriented framework have no inheritance mechanisms.

5.2.2 Separating Synchronisation and Functionality

One of the fundamental goals of DRAGOON is to improve the opportunities for reuse in the domain of embedded and large-scale software systems. Therefore, since the adoption of Ada-like concurrency features would have severly weakened the usefulness of inheritance, which is the most important reuse mechanism of object-oriented languages, DRAGOON instead employs a completely different strategy for achieving exclusion synchronisation in an object-oriented style.

The basic problem with the approaches described above is that the synchronisation conditions have to be explicitly embedded in the source code of classes by

the applications programmer, whether they are concentrated in a single process, or dispersed amongst the methods. There is no fundamental reason why this should be so, however. The synchronisation conditions applying to a particular method - determining *when* it may be executed - are essentially independent of its functionality - defining *what* it does.

Consider a class, for example, which models simple uni-buffer objects with a storage capacity of only one item, and which provides the conventional *PUT* and *GET* methods. There are at least three meaningful access protocols which can be used with instances of such a class [Atkinson et al, 90]. If *PUT* and *GET* were constrained to execute in strict alternation the buffer object would behave like a mailbox, with the consumer having to "*GET*" any produced item before the producer can "*PUT*" a new one. If, on the other hand, a simple mutual exclusion protocol was used, the consumer would be guaranteed only to receive the latest version of the produced items, since the producer would be able to overwrite unconsumed ones. Finally, another classic alternative is the readers/writers access protocol, in which the buffer behaves more like a "notice board" with concurrent *GET* operations, but exclusive *PUT*s. Even this basic protocol has a number of variants depending on the degree of fairness offered to writers wishing to perform a *PUT* in the face of an incessant stream of readers.

Since the approaches described above require the embedding of the access protocol within the class implementation, a different class would have to be defined for each of these protocols despite the fact that the functionality of the methods remains unchanged. This replication of code runs directly counter to the aims of reuse and incremental development which makes the object-oriented programming approach so popular. To overcome this problem, and the general conflict between inheritance and reuse, it is necessary to decouple the specification of the synchronisation constraints applying to the methods of a class from the definition of their functionality.

Kafura and Lee [89] have gone some way towards achieving this separation of concerns in the context of the Actor model with their notion of "behaviour abstraction". Although their language ACT++ has the notion of a centralised "object manager" to handle access to the internal state of an object, the rules determining which of the methods are "open" (i.e. may be activated) are specified in separate, explicitly named, "behaviours". This makes it possible to define subclasses which can extend the interface of the class and add the necessary additional "behaviours" by renaming those inherited.

Before describing the DRAGOON solution to this problem, the precise meaning of the term "behaviour" needs some clarification. In some contexts, such as the pure

Actor model, a "behaviour" refers to the "functionality" of possible responses to a message, while in others it is used to refer to the set of permissible operations in a given state. In DRAGOON, the *behaviour* of a class is defined as the synchronisation constraints defining its access protocol (i.e. the permitted interleaving of methods), and is therefore conceptually independent of the *functionality* of the class's methods.

5.3 Behavioural Inheritance

The basic philosophy of DRAGOON's approach to exclusion synchronisation is to permit the desired synchronisation constraints - the behaviour - to be added to any concrete functional class in a style analogous to the usual inheritance of implementations [Genolini et al, 89]. DRAGOON therefore introduces the concept of *behavioural inheritance*, which is a form of multiple inheritance in which one of the parents is a normal (but concrete), sequential class, and the other is an abstract description of a *behaviour*. The resulting class, termed a *behavioured* (or monitor) class, thus inherits the functional properties of one class in the normal way, and the behaviour of the other.

The abstract behaviour is modelled as a special kind of class, termed a *behavioural class*, which specifies synchronisation conditions using logic predicates over abstract method sets. These are not really classes in the true sense of the word, since such a class cannot be directly instantiated like others, but they are referred to as classes to signify that they may act as a parent in a form of inheritance relationship. Figure 5.1 illustrates the way a behavioured class is generated from a sequential and behavioural class using multiple inheritance. A behavioured class is a proper subclass of its unbehavioured parent.

5.3.1 History Functions and Deontic Operators

There are several possible ways of expressing abstract interleaving constraints on a set of operations, the most well known being path expressions [Campbell and Habermann, 74]. DRAGOON, however, has adopted an approach based on the use of logical assertions over method invocation histories, since these offer more expressive power than path expressions, and are in fact the basis of one technique for defining their semantics. These assertions are expressed in a form of *deontic* logic, which is an extension to first order predicate logic incorporating operators for expressing the deontic [1] concepts of "may" or "must".

The roots of deontic logics go back to Mally [von Wright, 80], who introduced the term in connection with work on capturing the concept of *willing*, but owe

[1] From the Oxford English Dictionary - "deontic: Pertaining to duty or obligation"

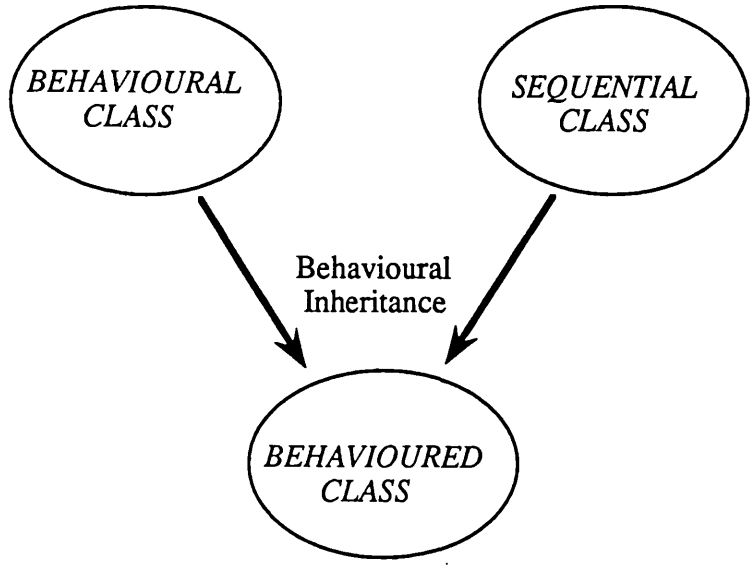


Figure 5.1: Behavioural Inheritance

their modern form largely to the work of von Wright [51]. The particular variant adopted in DRAGOON comes from a logic for requirements specification, called M[A]L [Khosla, 88], developed in the FOREST project [Goldsack et al, 89]. This introduces two deontic operators: "*obl*" - indicating that an agent is obliged to perform an operation before it does anything else, and "*per*" - indicating that an agent has permission to perform a particular operation. For the purposes of specifying abstract behaviours, however, only the "per" operator is required in DRAGOON, and is used to indicate the condition under which a given method may be executed, for example :-

per (op) ⇔ <boolean expression>

indicates that there is permission to activate a method "op" if, and only if, <boolean expression> holds. Rather than specify the conditions applying to individual operations, however, to increase the generality of behavioural classes the constraints are defined in terms of *sets* of abstract methods. This then permits the conditions specified by a single boolean expression to be applied to a number of methods which are equivalent as far as the behaviour is concerned. In practice, therefore, the deontic expressions have the form :-

per (*OPS*) ⇔ <boolean condition>

where "*OPS*" represents an abstract set of methods, and leads to the following informal interpretation - "there is permission to activate a method in the set "*OPS*"

if, and only if, <boolean expression> holds". "per (*OPS*)" is thus effectively defined as :-

per (*OPS*) $\equiv \forall$ op $\in$ *OPS*, per (op)

Three unbounded, and monotonically-increasing history functions are used to represent the number of times the methods of a particular set have been requested, activated or completed since system start-up :-

req (*OPS*)	=	the number of requests for execution of methods in the set *OPS*,
act (*OPS*)	=	the number of activations of methods in the set *OPS*,
fin (*OPS*)	=	the number of completions of methods in the set *OPS*.

Typically, however, permission to execute a method depends on the number of *currently* unserviced invocation requests (messages) or the number of *currently* active methods, so it is convenient to introduce special auxiliary functions representing these concepts and defined in terms of the primitive history functions :-

active (*OPS*)	=	act (*OPS*) - fin (*OPS*)
	=	number of currently active methods in the set *OPS*,
requested (*OPS*)	=	req (*OPS*) - act (*OPS*)
	=	number of currently outstanding calls for methods in the set *OPS*.

Like the primitive history functions they are defined in terms of, these auxiliary functions should return non-negative values.

5.3.2 Behavioural and Behavioured Classes

Using these five functions it is possible to define a large proportion of the behaviours that are likely to be needed in the development of concurrent systems. To illustrate how they are used, consider the possible behaviours that may be required in connection with the simple uni-buffer class mentioned above, which has a specification of the form :-

```
with ELEMENT;                                                    /5.1/
class UNI_BUFFER introduces
    procedure PUT (ITEM : in ELEMENT);
    procedure GET (ITEM : out ELEMENT);
end UNI_BUFFER;
```

The simplest behaviour that could be used with such a class would be straightforward mutual exclusion on the methods *PUT* and *GET*. In DRAGOON this behaviour is represented by the behavioural class *MUTEX* :-

```
behavioural class MUTEX is                                        /5.2/
ruled OPS;
where
    per (OPS) ⇔ active (OPS) = 0;
end MUTEX;
```

The "ruled" clause defines the names of the method sets that appear in the abstract behaviour. In this case, there need only be one set, called *OPS*, because as far as the behaviour is concerned there is no logical difference between the operations (i.e. methods) "ruled" by the behaviour. Informally the deontic assertions states that "there is permission to activate a method in the set *OPS* if, and only if, there are no members of this set which are currently active (i.e. executing)".

A uni-buffer class with a mutual exclusion access protocol can be defined simply by superimposing this behaviour on the methods of *UNI_BUFFER* using the behavioural inheritance mechanism :-

```
class MUTEX_UNI_BUFFER                                            /5.3/
    inherits UNI_BUFFER;        -- this class is a proper subclass of UNI_BUFFER.
    ruled by MUTEX;
where
    PUT, GET => OPS;            -- matches the set OPS with {PUT, GET}.
end MUTEX_UNI_BUFFER;
```

This class specifies that *MUTEX_UNI_BUFFER* is a subclass of *UNI_BUFFER* which is "ruled by" (or controlled by) the abstract behaviour specified in the behavioural class *MUTEX*. The "where" clause is the crucial part in which the concrete methods of the sequential class are mapped into the abstract method sets of the behavioural class. In this sense, behavioural inheritance can be viewed as the instantiation of the abstract (or generic) behaviour defined in terms of "formal" method parameters, on a particular set of "actual" methods provided by the sequential class. However, since in isolation a concrete (i.e. instantiated) behaviour is not a useful component of a DRAGOON system, it is the subclass relationship of the behavioured, or monitor, class to the sequential class which is emphasised.

In this particular case, the concrete methods *PUT* and *GET* are symmetric with respect to the behaviour, and therefore are mapped to the same set. This is not the case, however, with the "alternation" behaviour defined by the behavioural class :-

```
behavioural class ALTERNATION is                                    /5.4/
ruled FOP, SOP;                          -- F (irst)_OP and S (econd)_OP.
where
    per (FOP) ⇔ act (FOP) = fin (SOP);
    per (SOP) ⇔ fin (FOP) > act (SOP);
end ALTERNATION;
```

Informally, the deontic part of this class specifies that "a method in the set *FOP* may be activated if, and only if, for every activation of a *FOP* there has been a completion of a *SOP*, while *SOP* methods may be activated if, and only if, the number of completions of *FOP* methods exceeds activations of *SOP* methods". A version of *UNI_BUFFER* with an "alternation" access protocol is generated in precisely the same way as a "mutual exclusion" *UNI_BUFFER* except that on this occasion the methods *PUT* and *GET* are not symmetric with respect to the behaviour; the programmer must decide which of the concrete methods is to be assigned to the *FOB* set (i.e. which must be activated first) and which is to mapped onto the *SOB* set. Clearly, in this case, it is activations of the *PUT* method which must exceed activations of the *GET*, because an item must be placed into the buffer before it can be removed. The "where" clause of a behavioured subclass of *UNI_BUFFER* with this behaviour, therefore, would be of the form :-

```
where PUT => FOP, GET => SOP.
```

It is the programmer's responsibility to ensure that the assignments of concrete methods to abstract sets is consistent. Clearly, there must be at least one concrete method assigned to each abstract set, otherwise the conditions necessary to satisfy the deontic assertions would break-down.

Finally, the classic "readers/writers" protocol mentioned above, in which it is acceptable to permit concurrent reading of a data structure, but writing must be performed in strict exclusion, is defined by the following behavioural class.

```
behavioural class READERS_WRITERS is                                /5.5/
ruled ROP, WOP;                          -- R (eader)_OP and W (riter)_OP.
where
    per (WOP) ⇔ active (WOP) = active (ROP) = 0;
    per (ROP) ⇔ active (WOP) = 0;
end READERS_WRITERS;
```

The informal interpretation of the deontic assertions in this behavioural class is that "there is permission to execute a method in the set *WOP* if, and only if, there are no other active methods, while there is permission to activate a *ROP* method if,

and only if, there are no currently active *WOP* methods". There may be multiple *ROP* methods active concurrently, therefore, so the basic requirement of the "readers/writers" protocol is satisfied. This behaviour, however, is not "fair" to the *WOP* (writer) methods, because if the number of active *ROP* methods never drops to zero, a *WOP* will never gain access to the buffer. A fairer version of this access protocol can be generated simply by adding an extra condition to the deontic assertion for activation of *ROP* methods. A behavioural class defining a fair "readers/writers" protocol, *FAIR_READERS_WRITERS* say, would therefore be the same as the class *READERS_WRITERS* apart from the second deontic assertion which would be of the form :-

per (*ROP*) $\Leftrightarrow$ (active (*WOP*) = 0) and (req (*WOP*) = 0);

This extra condition means that "there is permission to execute a *ROP* method if, and only if, there are no currently active *WOP* methods *and* there are no currently unserviced requests for *WOP* methods". This ensures that "reader" objects calling *ROP* methods cannot block out "writers" indefinitely.

Whether the safe or unsafe form is used as an access protocol for *UNI_BUFFER*, the behavioural subclass concerned would have a "**where**" clause of the form :-

where *PUT* => *WOP*, *GET* => *ROP*.

There can be many simultaneous activations of the *GET* buffer, therefore, but only one exclusive activation of *PUT*.

When there are no active methods at all, both *WOP* and *ROP* methods may begin execution. Theoretically, this leads to a "race condition" because calls for *WOP* and *ROP* methods arriving at precisely the same moment may lead to the concurrent execution of the methods. In practice, however, evaluation of the execution conditions is a time-consuming and sequential process. DRAGOON, therefore, makes the assumption that the race condition is impossible in practice, and the hardware will select one of the calls for execution. This issue is discussed further in section 5.6.2.

5.3.3 The Exclusion Symbol

For all but the most trivial behaviour specifications, it is very common for methods to be permitted to execute only in complete exclusion with all other methods covered by the behavioural class. This is the case, for example, with the *WOP* method in the "readers/writer" behavioural classes described above. There is permission to activate a method in this set only if there are no other currently active methods of any kind. Using the notation described above, however, the only way to express

this requirement in the deontic expression is to explicitly state that the number of currently active instances of each formal method set mentioned in the "ruled" clause is zero. This causes no real problem in behavioural classes dealing with a maximum of two or three abstract method sets, but for behaviours involving more, it becomes extremely verbose, and unnecessarily obscures the meaning of the deontic expressions.

To counteract this problem a special composite symbol is introduced, called the *exclusion symbol*, which has precisely this meaning - that the current number of activations of all the method sets appearing in the behavioural class is zero (i.e. no methods are currently active). The symbol is represented by adjacent "greater than" and "less than" characters resembling an "X" shape - "(><)". Using this symbol, the conditions for execution of the *WOP* methods in the *READERS_WRITERS* behavioural class would simply have the form :-

per (*WOP*) ⇔ (><);

5.3.4 Behavioural Class Libraries

The behavioural inheritance mechanism outlined above overcomes the incompatibility of inheritance and exclusion synchronisation found in most other concurrent object-oriented languages because it completely decouples the expression of synchronisation constraints (behaviour) from the normal use of inheritance as a specialisation mechanism. A DRAGOON programmer designing a concurrent system requiring a particular "behavioured" class must first develop an "unbehavioured" class with the required functional properties. Only when this class is complete are the necessary synchronisation conditions added by means of the behavioural inheritance mechanism.

In a sense DRAGOON side-steps the root of the incompatibility problem, therefore, which is the *direct* specialisation of a behavioured class by inheritance. If a specialised version of such a class is required, the programmer must first specialise the original sequential version using the normal inheritance mechanism and then reimpose the necessary behaviour subsequently. DRAGOON behavioured classes can not be used as the parent of any other (apart from another special form of inheritance described in the next chapter) and so a class of this form represents the end of that branch of the (normal) inheritance hierarchy. It is not possible, for example, to generate a specialised version of the *MUTEX_UNI_BUFFER* class by using it as a parent. If such a class is required it is first necessary to define a specialised version of *UNI_BUFFER* and then reapply the synchronisation constraints using behavioural inheritance.

Booch Temporal Forms

Because the synchronisation conditions specified in behavioural classes are completely abstract, only a small number of them are required to handle most of the commonly required access protocols. The behaviours already introduced, for example, can be used to produce all of the important *temporal* forms defined by Booch to describe the properties of components in a concurrent environment. Booch identifies four different temporal forms :-

- **sequential** - the component is not protected in a concurrent environment.
- **guarded** - the component provides semaphore operations (e.g. signal and wait) which the clients must use to gain mutually exclusive access.
- **concurrent** - the component behaves like a simple monitor enforcing strict mutual exclusion on the exported operations. Operation requests are queued until they may be serviced safely.
- **multiple** - the component behaves like a monitor, but provides a more flexible readers/writers access protocol, which permits multiple simultaneous "reading" operations, but mutually exclusive (i.e. sequentialised) "writing" operations.

In the same way that the normal inheritance mechanism aids the definition of different spacial forms of an abstract component, as described in chapter 4, the behavioural inheritance mechanism of DRAGOON greatly simplifies the definition of these different temporal forms.

A sequential component corresponds directly to a sequential (i.e. unbehavioured) class in DRAGOON, which defines only the functionality of the methods exported. The guarded form employs an approach to exclusion synchronisation which is not normally recommended in DRAGOON designs, because it embodies all the risks of errors associated with the "orthogonal" approach identified above. Nevertheless, this form of component can be easily generated in DRAGOON by first introducing a subclass of the sequential form exporting two extra methods to act as the "signal" and "wait" semaphore operations, and then superimposing the required semaphore behaviour on this class using behavioural inheritance. An appropriate behaviour would be :-

```
per (SIGNAL) <=> act (SIGNAL) = fin (WAIT) and (active (OPS) = 0);
per (WAIT) <=> act (SIGNAL) > fin (WAIT) and (active (OPS) = 0);
per (OPS) <=> act (SIGNAL) > act (WAIT);
```

The most important component forms are the "concurrent" and "multiple" forms, which can be generated quite simply from the sequential form by using the "mutual exclusion", and "readers/writers" behaviours introduced above. The concurrent form, for example, would inherit the behaviour defined by *MUTEX*, with all the concrete methods of its sequential parent mapped onto the set *OPS*, while the multiple form would inherit one of the "readers/writers" behaviours (i.e. either the simple or fair version). In this case the designer must decide which of the exported methods are "readers" and which are "writers" and bind them to the appropriate method set names.

When Ada is used as the implementation language, each different component form has to be completely reimplemented as a different package. The situation is little better with the majority of concurrent object-oriented languages described above, since the different access protocols would have to be explicitly encoded in the body of the different forms. In DRAGOON, however, with only three behavioural classes, it is is possible to generate all the different "temporal" forms recognised by Booch without any replication of code.

Of course, the access protocols identified by Booch in his component taxonomy are not the only ones which may be needed in the development of DRAGOON systems. Nevertheless, the majority of behaviours likely to be required can be handled by a fairly small number (of the order of 10) commonly recurring schemes. The DRAGOON component library, therefore, will contain a standard set of of predefined behavioural classes which are intended to cover a wide range of the commonly required behaviours. In order to accommodate applications requiring a behaviour not available in the library, however, full DRAGOON compilation systems will permit programmers to define their own behavioural classes.

5.4 Active Objects

Having introduced the basic strategy for supporting concurrency in DRAGOON, it is now possible to examine the various features in more detail by considering the introduction of concurrency into the supermarket control system.

At the highest-level of system design, access to the data base by the two other principle object types in the system (i.e. check-out points and the central controller) essentially conforms to the "readers/writers" scenario. The central controller object "produces" new product descriptors and "writes" them to the data base object, while the numerous checkout point objects continuously "read" these descriptors in the database to determine product prices. Since such readers and writers are conveniently handled as autonomous processes, with their own independent execution

thread, in DRAGOON the central controller and checkout points are most naturally modelled as active objects, and the data base as a behavioured object, protecting its internal data from corruption.

The central control point provides the main interface to the system for human operators, and therefore is likely to be a fairly complex object aiming to make this interface as "friendly" as possible. So far as its interaction with the data base is concerned, however, details of how product descriptors are actually produced, and the handling of the interface, are unimportant. In order to focus on how active objects are defined in DRAGOON, therefore, the central controller object, for the moment, will be regarded as a simple producer, employing the library function *NEW_DESCRIPTOR* to create new descriptors, and then arranging for them to be added to the data base.

5.4.1 Threads

As in Modula-3 [Cardelli et al., 88] the independent execution thread which characterises an object as active is called the *thread* in DRAGOON, and may either be defined explicitly in the class body, or inherited from an ancestor. Classes which define a thread directly must indicate this fact to potential clients by means of the keyword "thread" in the "introduces" clause of the specification. Regarding the central controller as a simplified producer of product descriptors, therefore, it would be modelled by a class with a specification of the form :-

```
with DATA_BASE;                                                    /5.6/
class CENTRAL_CONTROLLER
    introduces
        procedure SHARE (SHARED_DB : in DATA_BASE);
        thread;
end CENTRAL_CONTROLLER;
```

The method *SHARE* is introduced as a means of providing instances of *CENTRAL_CONTROLLER* with a reference to a shared instance of *DATA_BASE* (4.23). Its body, therefore, merely assigns the actual parameter to one of the class's private instance variables :-

```
with PRODUCT_DESCRIPTOR, NEW_DESCRIPTOR;                           /5.7/
class body CENTRAL_CONTROLLER is
    DB : DATA_BASE;                    -- stores reference to the shared data base
    P : PRODUCT_DESCRIPTOR;
    procedure SHARE (SHARED_DB : in DATA_BASE) is
    begin
        DB := SHARED_DB;
```

```
        end;
    thread;
        loop
            P := NEW_DESCRIPTOR;
            DB.ADD(P);
        end loop;
    end CENTRAL_CONTROLLER;
```

As this body indicates, the thread of a class has a similar appearance to the initialisation part of an Ada package, except that it follows the keyword "thread" rather than "begin". Furthermore, like the initialisation part of a package, there can be a maximum of one thread per class. Unlike package initialisation code, however, which is executed when a package is elaborated, the thread of an object is only activated by an explicit invocation of the special method *START*, which is an implicit part of the interface of every active object. This approach also distinguishes DRAGOON from most other object-oriented languages adopting the active object approach, since in these the process part of an object is activated as soon as it is created. The reason for providing an alternative mechanism to this in DRAGOON is to enable the communication pattern of concurrent systems to be set up before the processes themselves are activated, as will be described more fully below.

When the *START* method is called by an external object, a new thread of control is generated to execute the sequence of statements in the thread part of the object. This will only have an extended lifetime (i.e. behave like a persistent active process) if it contains a loop, otherwise it will terminate on completion of the sequence of statements. In this respect, the DRAGOON approach is similar to the "asynchronous method execution" approach described above, since the thread is activated by a call to the method *START* and proceeds asynchronously until complete. A difference with this model, however, is that the special *START* method in DRAGOON may never have any parameters, and may only be called once for each instance of an active class. Subsequent invocations of the method result in the exception *START_ERROR* being raised.

In so far as their interaction with the data base is concerned, the checkout point objects will have a similar form to the central controller class - the difference being that they are *consumers* of product descriptors, rather than *producers*. Like *CENTRAL_CONTROLLER*, therefore, the class describing checkout points must export a method similar to *SHARE* by which the reference of the shared data base object can be made visible to the instances.

5.5 Agents

It is possible to define active objects which export no methods whatsoever and thus whose only function is to call other objects in the system. However such objects, which following the terminology of Booch [87] are called *actors*[2], are not used frequently, because to perform a useful role in a concurrent system a "calling" object must export at least one method like *SHARE* to receive reference to shared objects.

Conversely, of course, there are objects which possess no active thread, and which therefore are unable to issue calls spontaneously. Booch terms objects of this nature, *servers*, since they may only execute in response to method calls. This term is ambiguous in the context of object-oriented languages, however, where it is used to refer to any object (including active ones) which provides "services" to client objects by exporting a set of callable methods. A better term for describing objects which possesses no thread is *passive*. Clearly normal sequential objects in a non concurrent system fit the criteria for being passive, but when used in a concurrent system, such objects need to be protected by a suitable access protocol using the mechanisms of behavioural inheritance.

The final category of objects used in building concurrent systems are what Booch refers to as *agents*, that is, objects which both export methods for invocation by others, and possess their own thread to initiate calls. Since their methods may be called concurrently by other objects, they are likely to require an appropriate behaviour to control *external* access to the internal data. This does not, however, guarantee that the internal state of the object is protected from concurrent access, because like the methods the thread of the object has unrestricted access to its state variables. Although the behaviour is able to protect against the erroneous concurrent execution of *methods*, because it does not affect the thread it is unable to ensure this will not access an internal state variable at the same time as a method. In the case of instances of the *CENTRAL_CONTROLLER* (5.7) class, for example, invocation of the method *SHARE* while the thread is active could lead to interference and unpredicted results. The problem is avoided in this particular case because the thread should not be activated until the *SHARE* method has been called to initialise the external reference. In general it remains true, however, that the thread of an object is able to access its state variables concurrently with its methods.

The solution to this problem is to provide "internal" objects in danger of being concurrently accessed with their own behaviour. In other words, the burden of protection does not reside entirely with the encapsulating object, but also with

[2]Not to be confused with "actors" in the model of Agha and Hewit [Agha, 86]

its private objects. DRAGOON, in fact, places no restriction on the nature of private objects - they may be actors, passive servers and agents in any combination. Concurrency, therefore, may exist at all levels in the DRAGOON dependency hierarchy. Although an object may possess only one sequential thread, it may contain an unlimited number of private active objects, which essentially provide it with further internal threads. Even a passive object, which possesses no thread itself, may possess private objects which are active. DRAGOON also permits the temporary (instance) variables of methods to be active objects. Like an Ada subprogram which is the *master* of a task, a DRAGOON method does not terminate until the threads of all its private active objects (i.e. those it creates and activates) have terminated. Of course, a method may activate the threads of active objects of the encapsulating class instance without waiting on its termination.

This contrasts sharply with other concurrent object-oriented languages such as POOL, which enforce strict sequentiality within the encapsulation boundary of an object. In POOL the single execution thread (or body) that an object may possess is used both to make it "active" and to control external access in the style of Ada "**select**" statements. All its private (or internal) objects, therefore, reside in a completely sequential environment in which there is never more than one active thread of control. This approach has the advantage of simplifying the language's semantics, but the disadvantage of imposing an unnaturally flat concurrency structure on POOL systems. The DRAGOON approach, on the hand, makes the formal definition of semantics more difficult, but provides the full benefits of concurrency at all levels of the dependency hierarchy of a system. Although this may leave programmers with more leeway to produce erroneous programs which deadlock, it also provides scope for the definition of much more powerful programs.

5.5.1 Condition Synchronisation

Enabling agent objects to contain (private) behavioured objects is important because it provides the way of achieving condition synchronisation in DRAGOON. This is the form of synchronisation required when one process must wait for a signal from another before proceeding. It is possible to use a passive object, with the "alternation" behaviour of the class *ALTERNATION* (3.4), to act as the medium for synchronising two external objects in the style of a semaphore. The waiting process simply calls the second permitted operation (i.e. "*SOP*" in the behaviour) which only has permission to execute after the other process has called the first permitted operation (i.e. "FOP" in the behaviour). Although this is perfectly legal DRAGOON, it is a strongly discouraged approach because it relies on a third semaphore object in the style of the "orthogonal" approach described above, and suffers all the associated

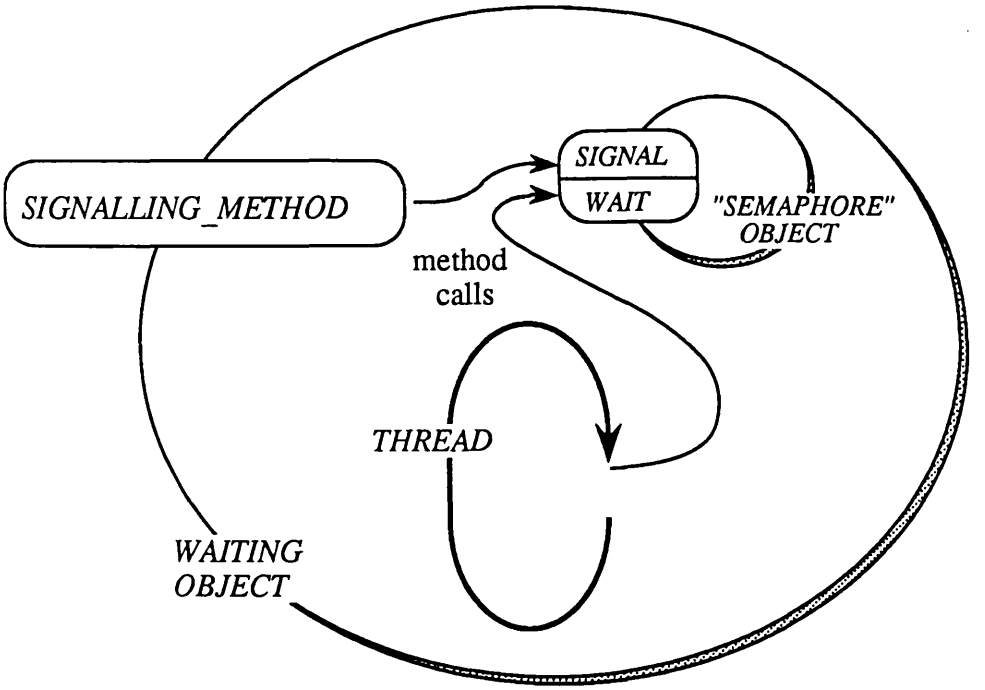


Figure 5.2: Condition Synchronisation

problems of successful use.

It is not actually the employment of a behavioured object as a semaphore that is the problem, but that this object appears at the same level of abstraction as the synchronising objects. This means that the responsibility for ensuring correct synchronisation lies with the component user rather than builder.

Since DRAGOON allows arbitrary nesting of behavioured and active objects, it is not necessary for the "semaphore" object to exist at the same level as the communicating parties - it may instead be a private object of the server. Condition synchronisation in DRAGOON, therefore, does in fact rely on a semaphore like object providing "alternation" behaviour on its methods, but this is private to the waiting object and so is invisible to the signaller. To transmit a signal to the waiting object the signaller simply calls a method in the waiter's interface, usually having no constraints on its execution. As illustrated in figure 5.2, this calls the appropriate method of the private semaphore object, which blocks the thread of the waiting object due its strict "alternation" behaviour.

This technique for condition synchronisation is just one possible use of nesting behavioured classes within each other. In principle there is no limit to the degree of nesting which is possible, but this facility must be used with great caution because the various behaviours can easily interfere. General speaking the behaviour of a class is determined not only by the behaviour it inherits directly by the behavioural inheritance mechanism - the *inherited* behaviour - but also by the behavioured con-

ferred upon its methods indirectly by their interaction with the behavioured classes of which it is a client - the *conferred* behaviour. If the programmer does not control the interaction of these two sources of behaviour carefully, they could easily interfere destructively. Suppose, for example, that an active object were required which waits for a particular signal and does not permit any of its own methods, other than that intended for receiving the signal, to be executed before the signal arrived. An appropriate behaviour for such an agent might be :-

per (*SIGNAL_OP*) ⇔ act (*OTHER_OP*) = act (*SIGNAL_OP*) = 0;
per (*OTHER_OP*) ⇔ act (*SIGNAL_OP*) > 0 and active (*OTHER_OP*) = 0;

By mapping the signal-receiving method of the agent to *SIGNAL_OP* and all the others to *OTHER_OP* in the behavioured class, this behaviour would ensure that no other method of the class could execute before the signal-receiving method, but after the signal was received, they would execute in mutual exclusion. If the body of the class is programmed correctly this is a perfectly acceptable behaviour to use in conjunction with the condition synchronisation technique. However, if a mistake is made in programming access to the semaphore object, for example, by the signal-receiving method calling the *WAIT* method of the semaphore, or alternatively the thread of the agent calling the *SIGNAL* method, the thread of the agent would be blocked indefinitely. Far from conforming to the inherited behaviour described above, therefore, the behaviour of the resulting class could best be described as "deadlock"

This is an extreme case of the kinds of interactions that occur between the inherited and conferred behaviour of an object. This can be used constructively to provide subtle behaviours that cannot be satisfactorily expressed by behavioural inheritance alone, but sometimes, of course, if wrongly used the resulting behaviour may not be that desired by the programmer.

Another potential source of problems in the use of behavioural inheritance arises if classes are permitted to invoke their own methods. Most of the behaviours superimposed on classes enforce total exclusion constraints (i.e. (><)) on at least some of the methods in their interface. If one method, therefore, called another of the classes methods with which it was constrained to be mutually exclusive then deadlock would obviously occur. To avoid this, it is necessary to ensure that mutual exclusive behaviour is never applied to a method making introspective method calls (i.e. calls to *SELF*). As mentioned in chapter 3, partly for this reason, and partly because the possibilities they offer can be easily achieved by other means, DRAGOON does not permit introspective method calls.

Even with this restriction on introspective method calls, there is still a great deal of leeway for programmer errors in the combination of inherited and conferred inheritance. This could be seen as a weakness of a language intended to reduce the scope for programmer error by such techniques as strong typing. However, it is important to realise that most of the applications of behavioural inheritance involve simple passive classes in which their is no risk of interaction between behaviours. Moreover, concurrency is an application area in which it is notoriously difficult to reduce the chance of errors without significantly reducing the flexibility and expressibility of the language. It is felt, therefore, that the DRAGOON approach provides a reasonable balance between flexibility and safety, providing a concise and easily understood mechanism for handling common synchronisation problems, but also providing features for handling more complex and unusual requirements in an elegant fashion.

5.6 Forwarder Objects

As the methods of the data base object will be invoked by numerous active objects, it must be protected by an appropriate behaviour. The most obvious candidate is the *FAIR_READERS_WRITERS* behaviour described above, since this ensures that the central control point (i.e. writer) will not be blocked out indefinitely by the considerably larger number of checkout points (i.e. readers). For this particular application, however, this behaviour is not entirely satisfactory either, because it is important that access to the data base by checkout points is as fast as possible. Using the *FAIR_READERS_WRITERS* behaviour to control access to the database will force the checkout point objects to wait for access to the data base whenever the central controller adds new product descriptors.

It is not actually necessary for the "writing" methods to be treated fairly with respect to the "reading" methods in this case, merely that the central control object does not have to wait for an extremely long period each time it wishes to add a new product descriptor. Moreover, since the load on the data base due to checkout points will fluctuate to some extent, it is highly unlikely that "writers" will be excluded indefinitely. All that is needed, therefore, is a mechanism for giving access priority to the checkout points, but ensuring that the central control is not blocked for long periods trying to add a new descriptor to the data base at a time of heavy usage by checkout points. Rather than attempting to define an overly complex behaviour to provide the desired balance, this requirement is best handled by introducing a buffering medium between the central control point and the database.

It would be possible, of course, to achieve this using a passive instance of the

class *QUEUE* (4.4) defined in the last chapter, into which the central control object "*PUT*"s product descriptors, and from which the data base object "*GET*"s them. However, to avoid making the data base into an active object, which this approach entails, the strategy which will be adopted instead is to use an *active* "forwarding" object which buffers product descriptors from the central controller until it can forward them to the data base. This is represented by the class :-

```
with DATA_BASE, PRODUCT_DESCRIPTOR;                                    /5.8/
class FORWARDER
    introduces
        procedure SHARE (SHARED_DB : in DATA_BASE);
        procedure ADD (P : in PRODUCT_DESCRIPTOR);
        thread;
end FORWARDER;

with QUEUE;
class body FORWARDER
    D_BASE : DATA_BASE;
    DESCRIPTOR : PRODUCT_DESCRIPTOR;
    BUFFER : QUEUE;                          -- stores reference to the shared data base
    procedure SHARE (SHARED_DB : in DATA_BASE) is
    begin
        D_BASE := SHARED_DB;
    end;
    procedure ADD (DESCRIPTOR : in PRODUCT_DESCRIPTOR) is
    begin
        BUFFER.ADD(DESCRIPTOR);
    end;
thread
    BUFFER.CREATE;
    loop
        DESCRIPTOR := BUFFER.POP;
        D_BASE.ADD(DESCRIPTOR);
    end loop;
end FORWARDER;
```

As illustrated in Figure 5.3 each forwarder has one private *QUEUE* object to store product descriptors waiting to be added to the data base. The method *ADD* is called by the central controller object to pass new product descriptors to the forwarder. As *FORWARDER* is not type-compatible with *DATA_BASE* (4.23), therefore, the class *CENTRAL_CONTROLLER* (5.7) should not actually be a client of *DATA_BASE*, as shown above, but rather of *FORWARDER* (i.e. the variable *DB* and formal method parameter *SHARED* would be of class type *FORWARDER* rather than *DATA_BASE*).

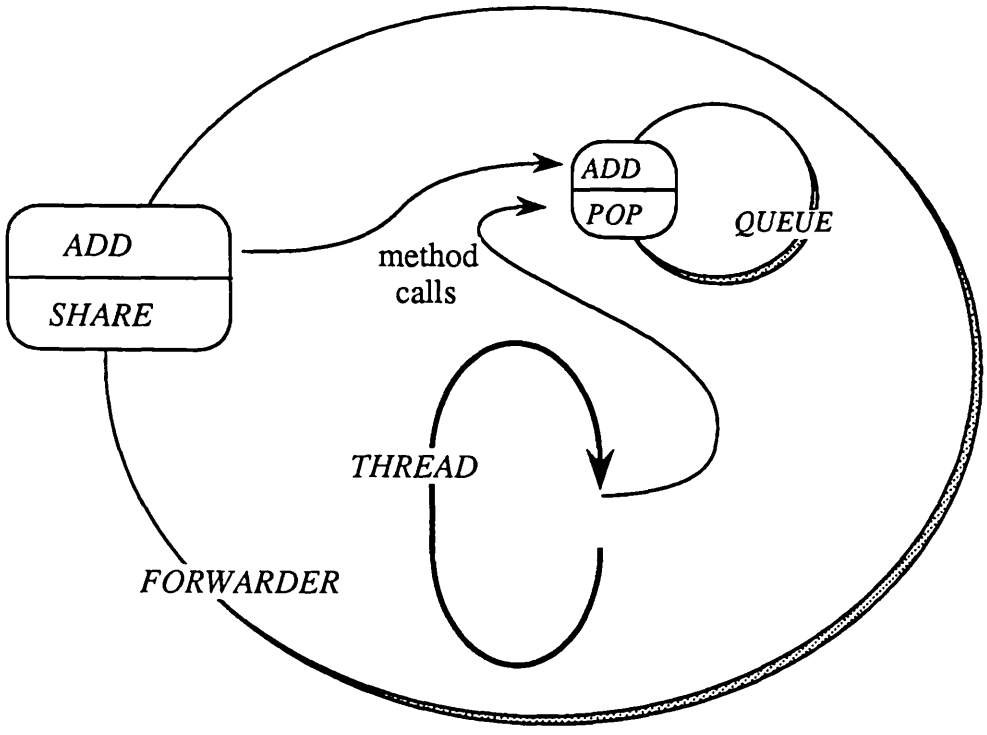


Figure 5.3: Structure of a forwarder object

Unless the buffer is full, the *ADD* method immediately adds the product descriptor to the buffer. The "forwarding" is performed by the thread of the forwarder object, which removes product descriptors from the buffer and *PUT*s them into the data base. As with *CENTRAL_CONTROLLER* objects, the reference to the data base is passed to forwarder objects by means of the method *SHARE*.

This design for forwarder objects is fairly straightforward, and can make use of the generic "queue" component developed in the previous chapter. The main difficulty is generating a version of the queue with the right kind of behaviour. All the previous behaviours superimposed on sequential classes have been describable in fairly concise and natural terms using the five basic functions, but the behaviour required for the buffer used in *FORWARDER* objects is more complicated because in addition to exclusion constraints it must handle access *conditions* related to the size of the buffer. It is only acceptable to execute the buffer's *PUT* method, for example, provided the buffer is not full, while it is only acceptable to execute its *GET* method provided it is not empty.

It is quite easy to express the required reentrancy conditions using the history functions defined earlier. For example, if the buffer had a capacity of twenty items, the appropriate synchronisation conditions could be specified by the behaviour :-

per (*PUT_OP*) $\Leftrightarrow$ (><) and act (*PUT_OP*) − fin (*GET_OP*) < 20;

per (*GET_OP*) $\Leftrightarrow$ (><) and fin (*PUT_OP*) − act (*GET_OP*) > 0;

In addition to the basic mutual exclusion requirement, this behaviour expresses the fact that "there is permission to execute a *PUT_OP* only if the number of completions of *PUT_OP*s does not exceed the number of activations of *GET_OP*s by 20 (i.e. the buffer is not full), while there is permission to execute a *GET_OP* only if the number of completions of *PUT_OP*s exceeds the number of activations of *GET_OP*s (i.e. it is not empty)". The problem with using this behaviour, however, is that because the size of the buffer it controls has had to be explicitly stated, it is only applicable to buffers of size 20. This, in turn, means that a different behavioural class would have to be defined for each possible buffer size, which is clearly incompatible with the general goal of providing a small number of behavioural classes capable of satisfying most applications.

5.6.1 Guarded Permissions

This is just one example of a more general limitation of the simple behavioural class approach described in section 5.3. The history functions, "act", "req" and "fin" , and their auxiliaries "active" and "requested", provide a simple record of part of the state of an object. The "act" and "fin" information, for example, can easily be recorded by integer variables that are updated each time a method starts and completes, while the information provided by "req" is available to processes in many other languages[3]. However, this information alone is very often not enough to define the complete state of an object, and hence further conditions must be satisfied to define the access protocol. The "fullness" of a bounded buffer, for example, cannot be determined by the method invocation histories alone, but depends crucially on the implementation of the class body - specifically, the size of the data structure storing the buffered items.

There are a wide range of classes of this nature in which the access conditions are determined by aspects of the implementation, as well as the method invocation history. To avoid having to define a specialised behavioural class for every class with particular implementation idiosynchrasies (e.g. the queue size in this example), it is necessary to extend the behavioural mechanism described earlier so that the synchronisation conditions can also be made to depend on aspects of the state of the behavioured class.

This is achieved in DRAGOON by permitting the value returned by boolean methods (i.e. functions) exported by a class to be used as one of the logical conditions

[3]In Ada, for example, the *COUNT* attribute can be used to establish the number of calls queued on a particular entry.

determining when a particular method may be executed. Instead of an explicit specification of the size of a queue in a "bounded buffer" behavioural class, therefore, the boolean expression defining the execution permission for the *PUT_OP* methods has a *guard* controlled by the state of the queue :-

```
behavioural class BOUNDED_BUFFER is                                    /5.9/
ruled PUT_OPS, GET_OPS, FULL_GUARD, OTHER_OPS;
    per (PUT_OPS) ⇔ (><)  and not (FULL_GUARD);
    per (GET_OPS) ⇔ (><)  and act (PUT_OPS) − act (GET_OPS) > 0;
    per (FULL_GUARD) ⇔ (><);
    per (OTHER_OPS) ⇔ (><);
end BOUNDED_BUFFER;
```

When this behavioural class is used as a parent of a behavioured class, the abstract methods appearing as guards in the deontic expressions are instantiated with a boolean method from the sequential parent. Methods used as guards are normal methods in the interface of a class and hence must also have the conditions under which they may execute specified in the behaviour. Any abstract method name used as a guard, therefore, must also appear in the "ruled" clause of the behavioural class, and must have a corresponding deontic expression specifying its execution conditions.

For a method to serve as a guard returning information about the current state of an object, the only behaviour which can usually sensibly be applied to it is strict mutual exclusion with all other methods in the interface (i.e. (><)), otherwise a concurrently executing method could change the state while it is being evaluated. Moreover, unlike other abstract method names in the behavioural class, which can be instantiated with a *set* of methods in a derived behavioured class, abstract methods used as guards may only have *one* method assigned to them, otherwise the execution permissions in which they appear would not be uniquely determined.

Using the behavioural class *GUARDED_BUFFER*, a behavioured version of the class *SIMPLE_QUEUE* could be defined by the following class :-

```
class GUARDED_QUEUE                                                    /5.10/
    inherits SIMPLE_QUEUE;
    ruled by BOUNDED_BUFFER;
where
    PUT => PUT_OPS, GET => GET_OPS, IS_FULL => FULL_GUARD,
                                                IS_EMPTY => OTHER_OPS;
end GUARDED_QUEUE
```

With these mappings of concrete methods to abstract set names the intuitive meaning of this behaviour is clear - "there is permission to execute a *PUT* method if, and

only if, there are no currently active methods and evaluation of the boolean method *IS_FULL* returns false (i.e. the queue is not full)". What is no so clear, however, is how this conceptual interpretation of "guarded permissions" relates to actual execution of the guard method. In other words, how and when are the guard methods evaluated?

5.6.2 Guard Evaluation Semantics

At first sight an obvious interpretation might appear to be that the guard method is evaluated whenever a new call is issued. For example, the method *IS_FULL* guarding invocation of the *PUT* method would be evaluated when a client object invokes the *PUT* method. This interpretation is acceptable on the occasions when the conditions for *PUT* are satisfied and an instance of the method may begin execution immediately, but it is not acceptable when the method does not have execution permission. In this case, the call to *PUT* needs to be placed on a queue of pending calls until the conditions for activation of the method are satisfied. *Logically* this occurs as soon another object in the system invokes a *GET* method to remove an object and create spare space in the queue. The question is, however, by what mechanism, and in which "thread of control" are the conditions for activation of the *PUT* method (i.e. the guard method) reevaluated ? Clearly the originators of queued calls to *PUT* are unable to do so, because by definition they are blocked. Adopting the interpretation that guards are evaluated only when the guarded method is *called*, therefore, is not a valid approach, because there is no mechanism for "waking-up" a queued call when the logical conditions tested by the guard become true.

As pointed out earlier in this chapter, in conventional languages using guards as the basis for synchronisation, monitor objects are regarded as possessing internal execution threads responsible for evaluating the guards at the appropriate moment. In Ada, for example, the guards controlling the acceptance of entry calls are evaluated when the task's thread of control enters a "**select**" statement encapsulating one or more "**accept**" statements. The evaluation of the guards, therefore, takes place in the thread of control of the "called" rather than "calling" task, and completely independently of calls joining the entry queues of the task. In fact, when a "**select** statement is evaluated, guards are evaluated for *all* the accept alternatives, regardless of whether there are any calls currently queued on the entries concerned. For a rendezvous with a guarded entry to begin, therefore, not only must the "called" task have executed an appropriate "*select*" statement, and the corresponding guard evaluated to true, but there must also be a currently queued call.

The required interpretation of guarded permissions must simulate the effect of evaluating the guards independently of method calls, but without relying explicitly

on the concept of an active thread of control within the monitor, because it is the avoidance of this concept in defining synchronisation constraints which has hitherto paid such dividends as far as reuse is concerned. To meet these requirements, the following interpretation is adopted for the evaluation of guard methods used to determine the execution permissions of others. *All* the guard methods[4] in the behavioured class are evaluated indivisibly in some undefined order :-

- once at object creation time,
- every time a non-guard method is completed (whether on not successfully).

The guard methods are *not*, therefore, evaluated whenever one of the corresponding guarded methods is *called.* On the contrary, except at object initialisation time, they are evaluated after the *completion* of every non-guard method, since the execution of such a method may have changed the state upon which the guard conditions depend. If, after the reevaluation of all the guards, a call which was previously queued now has permission to execute, it is activated by the behaviour implementation and serviced by execution of the appropriate method body.

Strictly speaking, therefore, it is a misnomer to call methods used to determine the execution permission of others as "guards", because they are not actually "called" at the time when the guard is checked to see if an external call may be accepted. More accurately stated, the purpose of the guard functions is to return the *value* to which the real "guards" are set. This value is then, in a sense, "remembered" by the behaviour implementation until needed to establish execution permissions, or until the completion of a non-guard method causes its reevaluation.

Because the guard values are only reevaluated after the completion of non-guard methods, it is important that the guard functions do not have any side-effect on the state of the object, otherwise this could invalidate the value returned by previously evaluated guard functions. This is a general requirement of functions and expressions used to provide guard values, whatever the language or execution model. It follows, also, that the execution conditions associated with guard functions when invoked directly by client objects will usually be total exclusion (i.e. (><)), or at least exclusion with respect to non-guard functions, but it is possible for a guard function to execute concurrently with a non-guard function if the latter manipulates a completely unrelated part of the object's state.

[4] those mapped onto abstract methods appearing on the right hand side of deontic expressions.

5.7 Building Concurrent Systems

Having discussed the main facilities provided by DRAGOON for constructing components of concurrent systems, it is now possible to illustrate how a complete system is actually created. In view of the measures taken to integrate the concurrency features with the object-oriented aspects of the language it is perhaps not surprising that this is achieved in a very similar way to the construction of a purely sequential system.

In non-concurrent object-oriented languages, a system (or program) is represented as a single class, called the *root* class in Eiffel, which generates all the other entities in the system as its private objects. Although DRAGOON has a different model for compiling and executing such system classes (described in the next chapter), a (non-distributed) concurrent system in DRAGOON is similarly defined as a single class which generates and encapsulates all the other objects. Such system objects are usually actors, therefore, which export no methods (other than the special *CREATE* and *START* operations).

Suppose, for the sake of argument, that a non-distributed version of the supermarket control system were required in which, initially at least, there were 10 checkout points. Using the components developed so far, a class generating such a system would be of the form :-

```
class SUPERMARKET;                                                    /5.11/

with CENTRAL_CONTROLLER, CHECKOUT_POINT, DATA_BASE;
class body SUPERMARKET is
    CC : CENTRAL_CONTROLLER;
    DB : DATA_BASE;
    F : FORWARDER;
    CP : array (1 .. 10) of CHECKOUT_POINTS;
thread
    CC.CREATE;                                          -- create objects
    DB.CREATE;
    F.CREATE;
    for I in 1 .. 10 loop
        CP(I).CREATE;
    end loop;
    CC.SHARE(F);
    F.SHARE(DB);                        -- setup required system configuration
    for I in 1 .. 10 loop
        CP(I).SHARE(DB);
    end loop;
```

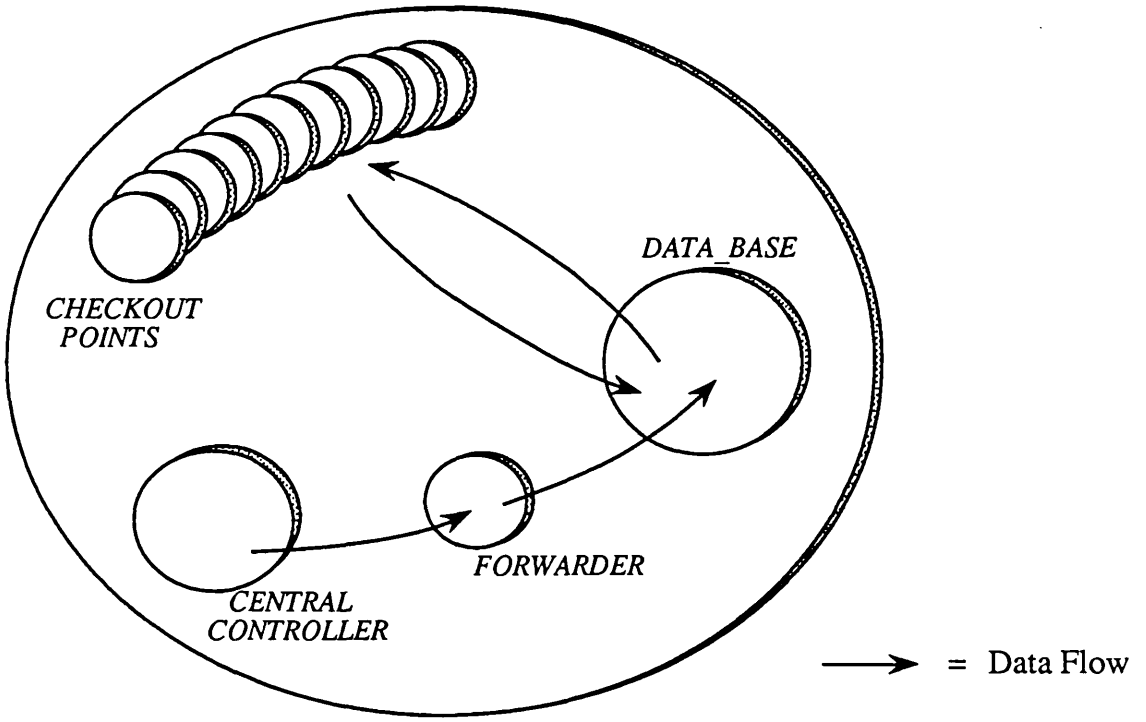


Figure 5.4: Non-distributed Supermarket Control System

```
    CC.START;                                    -- start objects
    F.START;
    for I in 1 .. 10 loop
        CP(I).START;
    end loop;
end SUPERMARKET;
```

This demonstrates one of the reasons why the thread of active objects is activated by a special *START* method, rather than at object creation time. It permits the required start-up configuration of the system to be established without concern for synchronisation problems while there is only one execution thread in the system. As pointed out above, if the *SHARE* method of a *CENTRAL_CONTROLLER* (5.7), *FORWARDER* (5.8) or *CHECKOUT_POINT* were called while the thread were active, untold errors could result. By enabling the required communication pattern to be established before the threads are activated, these problems are avoided.

The generation of an instance of the class *SUPERMARKET*, and subsequent activation of its thread by invocation of its *START* method will result in the generation of a supermarket control system as illustrated in Figure 5.4.

5.7.1 Reconfiguration

As defined above, the thread of the class *SUPERMARKET* behaves as a piece of initialisation code, serving only to generate the required objects in the system, set up the desired communication pattern, and activate the required execution threads. Once the thread of a *SUPERMARKET* object has performed these actions it terminates and plays no further role in execution of the system. This does not have to be the case, however. The thread of such an object could contain a non-terminating loop which provides it with an extended life-time, and permits it to continue regulating the configuration of the system.

Suppose, for example, that a version of the system was required in which a new *FORWARDER* (5.8) object is generated if the original instance becomes full, and hence the *CENTRAL_CONTROLLER* (5.7) blocked. This could be achieved by appending the thread definition above with code of the following form :-

```
        :
        loop
            <Wait until a new forwarder is needed>
            F.CREATE;
            CC.SHARE(F);
            F.START;
        end loop;
    end SUPERMARKET;
```

This illustrates the basic way in which the thread of a *SUPERMARKET* object could influence the configuration of the system at run-time. However, there are two important questions which must be answered before this technique can be made to work in practice. The first is how the thread is blocked until the condition under which conditions requiring the reconfiguration is detected, and will be considered further in the next chapter. The second is concerned with the fact that this "reconfiguring code" breaks one of main constraints mentioned above, which is the calling of the *SHARE* method of an object (in this case *CC*) after its thread has been activated. To overcome this problem it is necessary to define extra methods in the interface of *CENTRAL_CONTROLLER* which the *SUPERMARKET* must call to gain permission to perform the reconfiguration, and to inform the *CENTRAL_CONTROLLER* (5.7) when it has been completed.

Chapter 6

Distribution

The idea of superimposing special qualities on classes by an enhancement of the inheritance mechanism is used again to great advantage in DRAGOON for the construction of distributed systems. This chapter first outlines the special difficulties encountered in developing software for such systems, and the various ways in which the object-oriented paradigm has been applied to this problem. It then goes on to describe the particular approach followed in DRAGOON.

6.1 Loosely-Coupled Systems

The term "distributed system" is used to describe a large number of different systems, ranging from massive international networks of main-frames to small clusters of microprocessors communicating by shared memory. Those employed in the domain of embedded systems are *loosely-coupled* distributed processing systems, typically composed of a number of "embedded" microprocessors, and one or two larger *host* computers. The processors, or *nodes*, are "loosely-coupled" in the sense that they each possesses a separate memory unit, with the consequence that software segments executing on different nodes reside in disjoint address spaces. All communication between them, therefore, must ultimately be based on the exchange of message over the relatively narrow band-width network communication links.

More complex architectures are of course possible, in which several processors in an otherwise loosely-coupled network are "tightly-coupled" by shared memory. The arrangement of software amongst such machines, however, is more a problem for the compiler and run-time system than the applications programmer and language designer. For the present purposes it is convenient to regard such a "processor cluster" simply as another machine in the network.

Despite their extra complexity, distributed systems are becoming increasingly popular for the implementation of embedded systems because they offer a number

of significant advantages over their uni-processor counterparts [Sloman and Kramer, 87] :-

Performance. Being composed of multiple processors they are able to support the true parallel execution of concurrent processes, and therefore offer significant potential performance improvements for highly processor intensive applications.

Processor Locality. Processing power may be situated physically close to distributed devices, rather than concentrated in a large, centralised computer. "Long distance" communication can therefore be limited to the essential interaction between the devices, with all independent processing carried out locally.

Configuration Flexibility. Both the hardware and software architectures of the system may be modified periodically to reflect changes in the environment, or the availability of new devices or processors. These changes may be performed dynamically without disrupting those parts of the system not immediately affected by the reconfiguration.

Fault Tolerance. In contrast with a uni-processor system, the failure of hardware in a distributed system does not necessarily lead to the failure of the system as a whole. Damage can be limited to the particular node concerned, and the remaining software either reconfigured to manage without it, or substitute nodes brought in to compensate for the failure.

The extent to which these potential advantages can actually be realised in a distributed system, however, depends crucially on the nature of the language and design approach used to construct the application software. Languages poorly designed for this application domain can result in highly inflexible software structures which severely restrict the scope for dynamic reconfiguration and fault tolerance. Despite allusions to the contrary in the Ada Reference Manual [DoD, 83], it is now widely recognised that Ada is one such language whose modularity and concurrency mechanisms have been poorly combined for the purposes of flexible software distribution.

Before describing the features desirable for languages intended for programming distributed embedded systems, and how DRAGOON aims to improve the usability of Ada for this purpose, it is necessary to relate the goals of DRAGOON to that of other projects investigating the application of the object-oriented programming paradigm to the realm of distribution.

6.1.1 Object-Orientation and Distributed Systems

Many of the concepts of object-oriented programming are currently finding widespread application in loosely-coupled distributed systems. Object-orientation is probably most popular in the field of operating systems [Dasgupta, 86], [Snodgrass, 83], but projects such as Comandos [87] are using these ideas in the development of office automation systems, and Oscar [Shepherd et al, 88], for distributed robotics applications.

Although they address widely differing application domains, these projects have the common characteristic that they are essentially concerned with the development of object-oriented *environments* in which software is not only *developed* in an object-oriented style, but *executes* within the framework of an object-oriented run-time system. Consequently, they all adopt the philosophy of a "uniform" or "virtual" object space which hides the disjoint address spaces in the loosely-coupled network and permits all entities in the system, from the highest level application layers to the lowest level systems layers, to be modelled as objects

Implementing a fully general object-oriented environment on a distributed system is a notoriously difficult problem, however, because of the code sharing mechanisms that have to be supported [Wegner, 87], [Bennet, 87]. With a dynamic, (or reactive [Bennet, 87]), inheritance mechanism, such as that of Smalltalk, the functionality of a method may be inherited from objects situated on remote machines, and may be dynamically updated at any time. Unless large quantities of code are replicated on the separate sites, therefore, the execution of a method developed using dynamic inheritance may involve the execution of code dispersed throughout the network.

While construction of the run-time support and network communication mechanisms in an object-oriented style may well prove to be a suitable choice for DRAGOON systems, the "object-orientedness" of the final system is not a prime concern in DRAGOON. What is important is that the approach used should reinforce the benefits of distribution outlined above, should encourage the reuse of software, and should be sufficiently efficient to cope with the real-time demands of embedded systems. The motivation for adopting an object-oriented approach in DRAGOON, therefore, is not to apply the concepts uniformly in the construction of a distributed environment, but to bring the reuse advantages of object oriented programming to the development of distributed software. Consequently, DRAGOON is not concerned with supporting a uniform object model for its own ends, nor with supporting dynamic inheritance across the network. As will be explained below, however, the language does provide programmers with the full power of the inheritance mechanism for the development of distributable software components.

6.2 Programming Distributed Systems

As far as the development of embedded applications software is concerned the main factor distinguishing loosely-coupled distributed systems from uni-processor systems is the additional overhead of remote communication across the network, and the degree of control over this given to the applications programmer. Given the strict real-time constraints under which embedded systems must often execute, unless the likely delays in interprocessor communication are identified and minimised, the potential performance advantages of multiple processors can easily be outweighed by the extra overheads in remote communication. It is extremely important, therefore, that applications software is distributed judiciously amongst the processors in a network.

Traditional approaches for developing distributed software have fallen into two main camps [Atkinson, 88]. In one approach, no special linguistic support is provided for distribution, and the programmer must design a separate program for each machine in the network, expressing inter-program communication using the low-level communication facilities provided by the host operating system. A prerequisite of this approach, therefore, which may be conveniently termed the "operating system" approach, is full knowledge of the physical configuration of the target network and of the distribution of software amongst the processors. While it permits the most efficient implementation of inter-machine communication, since it provides access to the low-level primitives, this approach ties the application software to one particular system architecture, and limits its amenability to configuration change. Moreover, it reduces the reliability of the system because inter-node communication is not controlled by the typing rules of the language.

The second approach represents the other extreme in which the programmer requires no knowledge of the target configuration, and designs the applications software as if it were to execute on a single machine. It is the responsibility of the compiler or associated partitioning tools to split the software into components for distribution, and of the run-time system to support any required facilities across the network. This approach, therefore, which may be termed the "language oriented" approach, provides the opposite tradeoffs between efficiency and flexibility. Any program is clearly suitable for any configuration, since it contains no specific systems calls tying it to a particular architecture or communication system. However, since the software is designed without regard for potential distribution, its architecture may not lend itself well to efficient distribution.

6.2.1 Separation of Concerns

In applications where either efficiency or generality/reliability take precedence over all other considerations one of these approaches may provide a reasonable solution, but in the majority of situations a more equitable balance of properties is required. Recent approaches, therefore, have attempted to combine the advantages of the "operating system" and "language" oriented approaches, by providing a compromise between the total visibility and total transparency of the target configuration which they respectively assume. This is achieved by conceptually splitting the development of distributed application software into two distinct phases.

The functional modules for dispersal over the network, commonly called *virtual nodes*, are designed in the first *programming* (or *partitioning*) phase, while determination of the exact configuration of the target network, and the distribution of the virtual nodes upon it, is handled in the second *configuration* phase. By deferring configuration concerns until the second phase, the virtual nodes in a system can be designed without regard for the exact configuration of any particular target network, or arrangement of software amongst it. The resulting configuration independence of the application software provides a high degree of flexibility, not only with regard to the initial software architecture of a system, but also scope for dynamic configuration changes. Nevertheless, interactions between virtual nodes are still expressed in terms of high-level communications constructs of the language, and are thus type checkable by the compiler. This approach, therefore, permits much of the safety of the "language" oriented approach to gained in the first phase, and the efficiency of the "operating system" approach in the second.

Virtual nodes constitute the atomic units of distribution, whose contents cannot be separated across independent machines. Consequently, they can be regarded as abstractions of autonomous processing units, defining the granularity of distribution, and enabling applications software to be structured in a way that mirrors the loose-coupling of the underlying hardware. Programmers, therefore, must design the virtual nodes in a system in the knowledge that communication between them may ultimately take place by message passing over the network, and thus potentially carries a high overhead. The interior of a virtual node may be designed with complete freedom, however, since all its components are guaranteed to be colocated.

Many recent languages/environments designed specifically to support the development of distributed software explicitly embrace this approach. CONIC [Dulay et al , 87] for example, developed at Imperial College, provides a special linguistic construct, called a *task module* to model virtual nodes, and a distinct configuration language to describe the configuration of particular systems. Other linguistic constructs capturing the idea of virtual nodes include the guardian in Argus [Liskov,

82], the Resource in SR [Andrews and Olson, 86], and the sequence in MML [Boari et al, 84],

This approach is also proving to be one of the most popular techniques for overcoming the problems of distributions in Ada [Tedd et al, 84]. Both the DIADEM [Atkinson et al., 88] and Aspect [Hutcheon and Wellings, 87] projects having developed strategies for modelling virtual nodes in Ada, and for supporting Ada-like remote communication mechanisms.

6.2.2 Characteristics of Virtual Nodes

In view of the advantages of this approach for controlling communication overheads, while at the same allowing compile time checking of remote transactions, it would clearly be desirable if DRAGOON could pursue a similar strategy for software distribution. However, supporting the virtual node concept within the object-oriented framework is not as straightforward as might at first be thought. To see why this so, it is first necessary to identify the main properties required of virtual nodes, and the supporting communication system [Goldasck et al, 87], [Hutcheon and Wellings, 88].

- To perform their role as loosely-coupled node abstractions, virtual nodes must not communicate by direct access to shared data structures, but through well defined interfaces in terms of a message based protocol. Moreover, *all* state in the system should be encapsulated within virtual nodes.

- To model strongly cohesive modules, virtual nodes must tightly encapsulate their internal state, ensuring that no other virtual node can *ever* access this state directly. In particular, they should never export references to parts of their internal state, since this undermines the encapsulation boundary provided by the interface.

- To provide maximum flexibility in system configuration, and minimise the dependency between the programming and configuration phases, virtual nodes should be separately compilable library units, and should represent module *types* acting as a blue-print from which multiple instances can be instantiated.

- To permit dynamic changes in the communication pattern of the system, and incremental upgrades, instances of virtual nodes should not reference each other directly in transactions, but by means of an indirect naming scheme.

- To facilitate the generation of concurrent execution threads and take advantage of the presence of multiple processes, virtual nodes should be able to model

active processes with independent execution threads. This does not mean that all virtual nodes have to represent active processes, but merely that the capability for modelling such entities exists.

- To provide maximum flexibility in the allocation of virtual node instances to physical machines in the network, the distributed run-time system should support the same communication mechanisms for local (i.e. intra-processor) and remote (i.e. inter-processor) communication; a property known as *communication transparency* [Kramer and Magee, 85].

6.2.3 Suitability of Objects

At first sight classes/objects from the object-oriented programming world would seem ideal constructs for acting as virtual nodes in the partitioning of software for distribution. Classes are separately compilable "types" from which multiple instances (i.e. objects) can be instantiated, while objects not only communicate through well-defined interfaces by means of method calls, but name each other indirectly. Moreover, the concurrency extensions of languages like DRAGOON enable objects to have threads, and thus model autonomous activities. There is, however, one fundamental aspect of objects which obstructs their suitability for acting as virtual nodes - their failure to strongly encapsulate their internal state.

This might seem a surprising statement to make given the frequency with which the modularity and encapsulation properties of objects are emphasised. However, in all object-oriented languages that employ reference semantics for identifying objects, which is the majority (e.g. Smalltalk, Eiffel, Objective C, C++ and POOL), objects do *not* guarantee encapsulation of their state [Goldsack and Atkinson, 89]. Despite the fact that all communication between them is expressed in terms of method calls, the use of indirect references breaks down the encapsulation barriers and means that objects can directly access and manipulate the "internal" state of others.

This problem exists wherever indirect references are used to identify data structures. In Ada, for example, using access types to refer to tasks makes it possible for tasks instantiated within a package body to be directly visible to any other part of the program to which the access value is communicated, regardless of the scoping rules that otherwise apply. The problem is exacerbated in "pure" object-oriented languages which only provide objects, indirectly identified by references, as the means of representing state, because it is not possible for an object to communicate any information about its state to another object without also conceding permanent visibility.

Consider the *DATA_BASE* object in the supermarket control system, for exam-

ple. This contains a large number of *PRODUCT_DESCRIPTOR* objects which can be regarded as representing part of its state, and which no other object should be allowed to modify directly. The main role of this *DATA_BASE* object is to provide *CHECKOUT_POINT* objects with information about particular products stored in the corresponding descriptor. It is true that, initially, the checkout points can only gain access to the information stored in the product descriptors by calling the appropriate method in the data base's interface. However, in a pure object-oriented language the only way in which the data base can communicate the necessary information is by supplying the reference to the appropriate *PRODUCT_DESCRIPTOR* object. From this point onwards, the checkout point which receives the reference has direct and unrestricted access to the *PRODUCT_DESCRIPTOR* object concerned, without having to call methods of the data base. There is nothing the data base can do to stop the checkout point altering the state of the *PRODUCT_DESCRIPTOR* object at will (though it must do so, of course, via the product descriptor's methods).

6.2.4 Reference Free Communication

In "pure" object-oriented languages there is no way around this conflict between reference semantics and encapsulation. Objects in a system must communicate in order to do useful work, but the only way in which they can do so is by exchanging references, which gives the receiving object unlimited access to the "state" of the sender. This is not a serious problem in uni-processor environments since all references, in any case, reside in the same address space and there is no extra overhead in allowing any object to reference any other. The only drawback is from the modularity and reuse point of view in that a resuable component cannot protect itself from misuse when it exports a reference.

In a distributed environment, however, this lack of strong encapsulation is a serious drawback to the realisation of the "virtual node" based approach to distribution, and the control over remote communication overheads which this provides. All conventional languages/environments which have adopted this approach strictly forbid the exchange of references between virtual nodes [Liskov, 82] [Sloman and Kramer, 87] because this not only defeats the loose coupling of the virtual nodes by permitting remote data structures to be accessed as if they were local, but can easily lead to data corruption if care is not taken to distinguish remote references from local ones.

The only way of providing a "virtual node based" approach to distribution in an object-oriented language is to relax the "pure" philosophy, and support a mechanism other than reference passing by which objects may communicate with each other. One possible strategy is to define an alternative form of parameter passing

semantics, in which it is not *references* to objects which are passed in method calls, but *deep copies*. A "deep copy" of an object in Smalltalk or Eiffel provides a complete (recursive) copy of the object and all others that it references. Indeed, some languages, such as Matroshka [Crowl, 88] avoid references semantics altogether and use copy semantics for all communication. This approach would certainly avoid the passing of internal references over the network, and is feasible for simple objects with few references to others. However, objects do not have to become very large or complex before the overheads of implementing a copy over the network soon outweighs the advantages of avoiding remote references. It is not clear, either, what the semantics of copying active and behavioured objects should be in a concurrent environment.

The only practical way of avoiding communication by references in an object-oriented languages is provided by "hybrid" languages such as DRAGOON and Simula which support the use of conventional data types as well as class types. Objects in such languages are not forced to communicate by exchanging references, but may also exchange static data values. DRAGOON is particularly powerful in this respect, since it permits the full use of Ada's typing mechanism.

It is not desirable, however, to restrict communication between all objects to being by means of data values, since this would rule out the use of many of the most powerful features of object-oriented programming. Reference semantics is not only invaluable for exploiting the properties of polymorphism and dynamic binding, which are so important for reuse, but also for permitting the *sharing* of objects, as required for example, in the construction of the supermarket control system. It is essential to permit all these powerful features to be used in *building* distributable objects, since these are often fairly large modules composed of many other objects. All that is required to support the virtual node approach to distribution is to prohibit the exchange of state references between objects which are intended to be potentially distributable - that is, *virtual node* objects.

6.3 Virtual Node Objects

It is easy to state the general principal that potentially distributable virtual node objects should only communicate by exchanging data values, but without imposing a blanket reference-passing ban on all objects it is not so easy to see how this should be achieved. In traditional implementations of virtual nodes their modularity is guaranteed by requiring that they do not pass references to "internal", or "local", data structures to others. In the normal object-oriented model, however, the notion of one object being "internal" or "local", to another is not generally applicable,

since they exist in a completely uniform object space, and subject to the typing constraints, any instance variable may be made to refer to any object.

To specify more precisely what structures are permissible for virtual node objects, and how they should be designed, therefore, it is first necessary to provide a basis for deciding the "locality" of objects (i.e. whether they are "internal" or "external" to others) by identifying the different purposes for which references may be used in a normal, non-distributed object-oriented program.

During the execution of an object-oriented program, an object may bring another into existence by invoking the *CREATE* method through one of its instance variables. A reference so created will be termed an *ingenerate reference* of the *creator* object - that is, the object which owns the instance variable and invokes the *CREATE* "method". Not all references used by an object, however, are ingenerate. Others may be *imported* from an anonymous client object as an argument of a method call, or alternatively from a server object identified by a previously imported reference.

During the execution of a program, therefore, the references contained by the instance variables of an object may be divided into two distinct categories :-

Ingenerate References - created explicitly by the object on invocation of the *CREATE* operation,

Imported References - received as method call arguments from other objects not identified by ingenerate references.

References of the first kind are used in two distinct scenarios. Either the ingenerate reference :-

- is *never* passed to an object identified by an imported reference. In this situation, the referenced object may legitimately be regarded as a completely *private* object "owned" by its creator, and used as a *component* in its construction. [Goldsack and Atkinson, 89]. An example of such an object is the *QUEUE* component of a *FORWARDER* (5.8) object. This is an object used merely in the construction of *FORWARDER* objects, and need never be referenced by any other object in the system. Used in this way, a reference defines the analogue of the "include" relationship in HOOD [87].

- *is* deliberately passed to objects identified by imported references. This scenario is used commonly in producer/consumer situations in which a producer builds objects for use by another. An example are the *PRODUCT_DESCRIPTOR* (3.4) objects created by the *CENTRAL_CONTROLLER* (5.6). Immediately after generating such an object, and filling its fields through interaction with

the user, the *CENTRAL_CONTROLLER* object passes its reference to a *FORWARDER* (5.8) object before proceeding to generate another. The *PRODUCT_DESCRIPTOR* objects of the *CENTRAL_CONTROLLER* cannot, therefore, be regarded as private components, because they are designed specifically to have their reference exported. It is probably better to think of the whole *PRODUCT_DESCRIPTOR* object, rather than merely the reference, as being passed in the *FORWARD* transaction, since the *CENTRAL_CONTROLLER* immediately creates a new *PRODUCT_DESCRIPTOR* object and relinquishes visibility of the old one.

Imported references of the second kind are important to enable objects to share access to other objects in the system, and are thus essential for configuring useful concurrent systems such as that described by *SUPERMARKET* (5.11) in the last chapter. *FORWARDER* (5.8) and *CHECKOUT_POINT* objects, for example, possess instance variables which contain imported references to facilitate their sharing of the *DATA_BASE* (4.23) object. Such instance variables are analogous to the *ports* of languages like CONIC, since they act as an indirect name for an external service. Imported references of this form define the analogue of the "use" relationship in HOOD [87].

Having identified these three different ways of using references in normal object-oriented programming, it is possible to state more precisely what criteria an object must satisfy in order to qualify as a virtual node :-

- A virtual node object should never pass an ingenerate reference to an object identified by an imported reference.

Essentially this means that scenarios of the second kind illustrated above, and exemplified by the exchange of *PRODUCT_DESCRIPTOR* objects in the supermarket control system, are not permissible for virtual node objects. They may possess and manipulate private objects in a completely unconstrained fashion, and may possess and exchange references to other virtual node objects, but they are *not* permitted to exchange ingenerate references.

Instance Variables Categories

Although "typed" object-oriented languages, such as DRAGOON and Eiffel, permit instance variables to refer only to subclasses of their dynamic type, no restrictions are placed on the "locality" of the objects to which they refer. In other words, during the execution of a program, any instance variable of an object can alternatively refer to ingenerate or imported references, because apart from the typing rules there

are no constraints on assignments between instance variables. Ensuring that the criteria laid down above are satisfied, therefore, is a difficult "booking keeping" task, analogous to the problems of programming in untyped languages.

To overcome this problem it is helpful when building virtual node objects to firmly differentiate instance variables referring to the different kinds of object identified above, and ensure that no assignments take place between them. Since imported references are analogous to ports in traditional distributed programming languages, as mentioned above, it seems sensible to call instance variable designed to contain such references as *port* variables.

Private objects, on the other hand, constitute part of the state of the creator object, and so it seems reasonable, therefore, to term instance variables intended to refer to such objects as *state variables.* In the same way that "port" variables may contain only imported references, "state" variables may refer only to ingenerate ones.

Method Categories

Prohibiting assignments between "port" variables and "state" variables is not actually sufficient to ensure that they will always contain the intended kind of reference however. References may be assigned to instance variables not only by direct assignment statements, but also via method call parameters. For complete security, therefore, it is also necessary to identify two different categories of methods.

Since it is only methods which have parameters of a reference type (i.e. a class or dynamic data type) that may cause the illegal passing of references, such methods will be termed *unsafe.* Methods with no parameters, or merely parameters of static data types will correspondingly be termed *safe.*

The final constraints placed on the structure of virtual node objects to ensure a strict separation of the different kinds of references are that the unsafe methods of a class may never manipulate its *state* variables, and conversely, *state* variables may never be supplied as method call parameters to objects referenced by *port* variables. In effect, the only action an unsafe method of a virtual node object may perform with arguments of reference types is to assign them to the object's port variables. This is illustrated by the *ASSIGN_DATA_BASE* method of the class *FORWARDER* (5.8). Such methods, therefore, serve as the mechanism for controlling the "configuration" of a system of interacting distributed objects, assigning imported references to virtual nodes' port variables, and are consequently also termed *configuration* methods.

6.3.1 Virtual Node Construction Rules

The guidelines for the construction of virtual node objects can be summarised as follows :-

- Instance variables should be divided into two categories - "state" variables and "port" variables.
- Methods should be divided into two categories - "safe" methods which possess no parameters of a reference type, and "unsafe" or "configuration" methods which do.
- There should be no assignments between state and port variables - these should remain completely independent.
- The "in"parameters of an unsafe method should never be assigned to the object's *state* variables. This is guaranteed by ensuring that the body of an unsafe method does not manipulate (i.e. name) the class's state variables or internal procedures.
- A reference stored in a *state* variable should not be supplied as the actual parameter of a method call issued through a *port* variable.

The basic goal of these rules is to ensure that virtual node objects do not communicate by means of reference passing. What they essentially provide, therefore, is a concrete notion of "locality" in an object-oriented system. Private objects, identified by state variables, can reasonably be regarded as "internal" to, or encapsulated within, a virtual node object, since the rules guarantee that no other virtual node will be able to reference them. Similarly, objects referenced by a port variable, and generated elsewhere, can reasonably be regarded as "external" objects. These construction rules, therefore, serve to enforce the tight encapsulation boundaries which the reference semantics of object-oriented languages undermine.

It is stressed, however, that the only objects which need such strong encapsulation boundaries, and thus need to be designed according to these rules, are those intended to be potentially distributable (i.e. virtual nodes). Objects intended to serve only as components in the construction of others may be designed in the usual way with no restrictions. Moreover, although virtual node objects are potentially distributable, this does prohibit them from also being used as normal components in the construction of other objects.

At present, there are no syntactic constructs available in DRAGOON to enforce these rules, they are merely strongly recommended guidelines of the design method.

If an attempt is made to export an ingenerate reference at run-time, however, the exception *REMOTE_ERROR* is raised.

These constraints on the structure of virtual nodes do sacrifice some of the power of the object-oriented programming paradigm across virtual node boundaries. In particular, it is not possible to take advantage of polymorphism over a network. However, this cannot be avoided if the overheads of network communication are to be contained. The important point is that inheritance can be used without restriction in the construction of virtual nodes, since in DRAGOON it is a static mechanism in which all relationships are resolved at compile time.

6.4 Virtual Nodes in the Supermarket System

Although it is certainly practical to control the devices of the supermarket control system from a single program such as an instance of *SUPERMARKET* (5.11), this system clearly lends itself well to implementation as a distributed system, with the central controller, data base, checkout points and possibly forwarders residing on separate machines. However, as currently defined, none of these classes satisfies the criteria outlined above as necessary to serve as virtual nodes for distribution, because of the way in which they exchange references to product descriptor objects. To qualify as virtual nodes, objects are not permitted to communicate by passing references, or to contain references to other objects which are not virtual nodes. It is not possible, therefore, to build a distributed version of the supermarket system from the current versions of the classes. For this to be possible the potentially distributed classes should have been designed from the beginning to conform to the virtual node construction rules.

Two main changes must be made to the current implementation of the classes in the supermarket system to make them conform to these rules. The first is to arrange for the information stored in product descriptor objects to be transferred around the system as static data values, rather than as object references. The second is to change the way in which the "fuse rating" associated with electrical descriptors is represented. At present the required rating is specified by means of a reference to the descriptor of the actual fuse product that is needed. However, this is not acceptable when the product descriptor concerned may reside on a separate machine. This property of electrical products, therefore, needs to be represented as an attribute of a simple enumeration type like all the others.

Consider the problem of adding new *PRODUCT_DESCRIPTOR* (3.4) objects to the data base. If this is to be a virtual node, it is not possible to do this by passing a *PRODUCT_DESCRIPTOR* reference as a parameter of a single *ADD* method. Instead

the attributes of the objects must be given to the data base individually as data values. There are two ways in which this can be achieved; either by providing a separate method for each attribute, or by a single method with an argument for each attribute. Even though the overloading facility of DRAGOON permits all the different methods required in the first approach to be given the same name, the second is clearly preferable since it is simpler and carries fewer invocation overheads.

The difficulty in the second approach is in coping with the polymorphism that was possible in object-oriented languages. In the original version of the class *DATA_BASE* (4.23) based on the use of object references, a single *ADD* method is sufficient to insert all the different kinds of product descriptor into the data base, because the *ELECTRICAL_DESCRIPTOR* (3.8) and *PERISHABLE_DESCRIPTOR* (3.6) are subclasses of *PRODUCT_DESCRIPTOR*. One solution, of course, is to define a separate method for each different type of product descriptor, but this again is unnecessarily complicated because the DRAGOON facility for providing method arguments with default values permits the original polymorphism to be simulated.

Since the bar-code, stock-level, price and label attributes are common to all the different types of descriptor these are treated as normal parameters of the method without default values. The arguments corresponding to attributes which are not common, on the other hand, such as the sell-by-date and fuse-rating, are provided with default value. This default value indicates to the body of the method that the attribute corresponding to the argument is "undefined" - that is, not present in the descriptor type denoted by the current call.

Because the method argument must have the same type as the corresponding attribute, this type must contain a value capable of representing this "undefined" state. It is possible in some cases for a normal value of the type to satisfy this, such as a past date, for example. In general, however, it is preferable to make explicit provision for this value in the definition of attribute types. The package *SIMPLE_CALENDAR* (3.3), for example, which defines the type of the date attribute of perishable products (as a private data type) could provide an additional function, *UNDEFINED* say, which would return this special value. Similarly, the enumeration type which is now needed to represent the fuse- rating attribute of electrical products would contain a special literal for this value :-

```
package FUSE is                                                    /6.1/
    type RATING is (THREE_AMP, FIVE_AMP, THIRTEEN_AMP, UNDEFINED);
end FUSE;
```

A new version of the data base conforming to the virtual node construction rules, *VN_DATA_BASE* say, would have an *ADD* method of the form :-

```
procedure ADD (PRICE : in PRODUCT.VALUE;
               BAR_CODE : in PRODUCT.CODE_NUMBER;
               STOCK_LEVEL : in PRODUCT.STOCK_NUMBER;
               LABEL : in PRODUCT.LABEL;
               SELL_BY_DATE : in SIMPLE_CALENDAR.DATE :=
                                    SIMPLE_CALENDAR.UNDEFINED;
               REQUIRED_FUSE : in FUSE.RATING := UNDEFINED);
```

Using these default values it is possible for the body of the *ADD* method to "rebuild" a *PRODUCT_DESCRIPTOR* object of the same form as that whose referenced would have been supplied directly in the original version.

```
        :
    PROD : PRODUCT_DESCRIPTOR;                                  /6.2/
    PERISH : PERISHABLE_DESCRIPTOR;
    ELEC : ELEC_DESCRIPTOR;
        :
begin
    if SELL_BY_DATE = UNDEFINED then
        if REQUIRED_FUSE = UNDEFINED then
            PROD.CREATE;
        else
            ELEC.CREATE;
            ELEC.SET_FUSE_TO(REQUIRED_FUSE);
            PROD := ELEC;
        end if;
    else
        PERISH.CREATE;
        PERISH.SET_SELL_BY_TO(SELL_BY_DATE);
        PROD := PERISH;
    end if;
    PROD.SET_BAR_CODE_TO(BAR_CODE);
    PROD.SET_PRICE_TO(PRICE);
    PROD.SET_STOCK_TO(STOCK_LEVEL);
    PROD.SET_LABEL_TO(LABEL);
        :
```

Clearly, it is possibly for the *FORWARDER* (5.8) responsible for inserting new product descriptors into the data base to "decompose" a *PRODUCT_DESCRIPTOR* (3.4) object in a precisely analogous, but reverse fashion, and to supply its individual attributes as actual parameters of this *ADD* method. Since the methods of virtual node classes can "decompose" and "reproduce" objects in this way, the fact that it is not actually references but data values that are communicated can be hidden from the rest of the implementation. Essentially, the programmer is explicitly coding up the "deep copy" semantics of parameter passing identified as one possible strategy for avoiding the exchange of information by reference passing.

Although it would certainly be possible for the DRAGOON implementation to generate this code implicitly for simple objects such as product descriptors, in DRAGOON the programmer is required to program this up explicitly, since it then ensures that he/she is aware of the overheads involved, and can be sure that a sensibly sized object is being transferred in a single transaction.

6.5 Executable Objects

The previous sections have described the structure of potentially distributable virtual nodes in DRAGOON, and how the classes in the supermarket system should have been designed to satisfy the necessary conditions. Conformance to these rules, however, does not automatically mean that instances of the class can be distributed freely amongst the nodes of a network. Objects which are able to execute independently on network machines possess important properties which are not captured in the virtual node class definition alone. This section describes the way in which the appropriate kind of objects can be created in DRAGOON.

There are two basic models by which high level computer software may be executed. The code can either be *interpreted*, in which case small segments, usually individual lines, are executed directly by the processing system, and put into immediate effect; or the code can be *compiled*, in which case it must first be translated into a different form before execution may take place. Traditionally these have tended to be regarded as alternatives, since the "compilation" process usually translates the "source" code directly into the machine code of the target computer. However, execution of machine code at this level is essentially just interpretation performed directly by the hardware. In this sense, therefore, it can be argued that there is an interpretation phase in all execution strategies. The only difference between the various approaches is how much of the required processing is performed *before* execution (i.e. interpretation) time in one or more "compilation" steps. Languages which are primarily interested in run-time efficiency, such as those intended for the development of industrial and commercial software, and for embedded systems, tend to maximise the amount of work performed in the "compilation" phase by translating directly into the target machine code, and are thus usually termed "compiled" languages.

The exact meaning attached to the term "compilation", however, differs somewhat depending on the precise nature of the language and development system used. In languages such as Ada and Modula, which permit application software to be developed as a set of separate modules, "compilation" describes the act of submitting a new module to the "compilation system" so that, if error free, it may be added

to the existing set of accepted modules. Compilation in this sense, therefore, is largely concerned with type checking and library management. A certain amount of translation may take place at this stage, but the production of an executable software module requires at least one additional step, typically termed "linking", which collects together the relevant components, and translates them into an executable form. For the present discussion, the exact mechanics of translating source modules into executable form are not important. What is is important is that in so called "compiled" languages, such as DRAGOON, at least one non-trivial (i.e. time-consuming) translation step is required before code execution is possible.

The Main Program

Although a language may permit many different types of module to be "compiled", in the above sense, most languages only recognise one construct, usually termed the "*main program*", as being translatable by the compilation system into an *executable* module. Pascal, for example, has an explicit "program" construct providing the scope for all other declarations, while in Ada any library *subprogram* may be used as a main program. Compiled object-oriented languages such as Eiffel, on the other hand, usually permit library classes to serve as main programs.

In languages such as these which were not designed specifically for the purpose of distribution, the constructs used to represent main programs were chosen under the assumption that a *single* program provides the functionality of an entire system - in other words, that every program executes completely independently of others. In the case of Ada, which had support for concurrency as one of its main design aims, this assumption required the designers to provide elaborate concurrency mechanisms that are orthogonal to the "main program" concept. A single Ada program, therefore, may contain multiple, concurrent tasks executing under the control of a single program's multi-tasking kernel.

This "single-program" assumption is not valid, however, in distributed systems, where the software modules allocated to individual nodes must clearly be "compiled" modules capable of being executed [Atkinson and Di Maio, 89]. In operating system terminology, such modules are often termed "heavyweight" processes, since they carry with them all the special run-time support required by features of the language. In contrast, processes (or objects) which exist within the bounds of such a heavyweight process, such as tasks in an Ada program, are termed "lightweight" processes since they carry no dedicated run-time support with them, but execute under the control of the encapsulating heavyweight process.

Because a distributed system is necessarily composed of multiple, interacting *heavyweight* processes (or objects) the "single main program" assumption lies at the

heart of Ada's incompatibility with distribution. The language does not recognise tasks, the in-built units of concurrency, as being executable outside the context of a main program (in fact, it does not even permit them to be library units), and consequently they cannot easily be used as units of distribution. The only way of doing so depends on viewing the entire distributed application software as a single "distributed" Ada program [Burns et al., 85], but as well as being difficult to implement, this conflicts with the principle of separating the programming and configuration phases outlined above.

Ada main programs, on the other hand, were designed to execute independently and are not therefore supported by mechanisms of the language enabling separate programs to communicate. This can only be achieved by using the communication facilities of the host operating system in the style of the "operating system" approach outlined above. A programmer adopting this approach, therefore, must in general handle two levels of concurrency; one at the level of "lightweight" tasks "internal" to an Ada program, and one at the level of " heavyweight" processes, communicating using facilities "external" to a program [Atkinson et al, 90]. This situation runs directly against the goals of reuse, since a component designed according to one concurrency model cannot later be reused in the other.

Consider, the supermarket control system, for example. To build a version of the system in Ada to execute within the bounds of a single program, the main components (i.e. data base, central controller and checkout points) would have to be implemented as (lightweight) tasks. To build a distributable version, however, they would have to be implemented as Ada programs communicating using the message passing facilities of the target system. Not only is the programmer therefore forced to handle two different levels of concurrency, but must decide from the earliest stage of system design which model to pursue. This is much worse than having to decide whether a component should be modelled as a virtual node or not, since in this case the programmer must only decide whether a class is to be a *potentially* distributable object. Deciding whether to structure a component according to the "internal" or external" concurrency model, however, completely rules out its use in the other way.

Ada is by no means the only language which provides a different model for heavyweight and lightweight processes. Most of the conventional languages designed without regard for distribution, but with a view to concurrency, such as Modula-2, Simula and CLU [Liskov et al., 81] fall into the same category. Even languages designed to support distribution, such as Argus [Liskov, 82] and EPL [Almes et al., 85], provide different models of concurrency for lightweight and heavyweight processes (respectively "CLU Clusters" and "Euclid data structures" versus "Guardians" and "Eden objects") [Black et al., 87].

6.5.1 "Heavyweight" and "Lightweight" Objects

To enable software components to be reused as either "lightweight" or "heavyweight" objects irrespective of there location or encapsulation within other components, it is necessary to support a uniform concurrency model and communications mechanism. This is demonstrated by many of the recently defined language/environments for developing distributed software such as CONIC and Emerald. In CONIC, for example, the task module is not only the unit of encapsulation and concurrency for modelling distributable virtual nodes, but can also be used to describe concurrency within virtual nodes. Projects such as DIADEM have attempted to overcome Ada's lack of symmetry in this respect by using the notion of virtual nodes to provide a uniform approach to concurrency. One of the main aspects of the DIADEM work was the implementation of a "remote rendezvous mechanism" enabling distributed virtual nodes to communicate using the same rendezvous mechanism as virtual nodes grouped together on a single machine.

Providing a uniform concurrency model in this way, however, creates the new problem of distinguishing between heavyweight and lightweight versions of a component in environments where they coexist. Emerald essentially avoids the problem because all the objects in a system are essentially "heavyweight", any object in the system being transferrable to any machine. As pointed out above however, this "uniform object space" model carries with it too many overheads to be generally acceptable in embedded system. The CONIC and DIADEM systems in a sense avoid the problem as well, because they do not handle the generation of "heavyweight" components within the language used to design their lightweight counterparts. As in most development environments, the task of compiling and linking components to translate them into an executable form is handled by the host operating system in response to command line instructions.

In these systems, therefore, it is necessary to leave the language environment to produce an executable version of a particular component and return again to configure a system. Not only does this lower their "user friendliness", but makes it difficult to distinguish between the many different versions of a component that may be produced.

6.5.2 Execution Support Classes

In DRAGOON all the different kinds of components used in a system, whether executable forms for distribution over a network, or non-executable forms for use in the construction of others, are represented in the normal inheritance hierarchy. This is achieved by using a special form of multiple inheritance to distinguish heavyweight

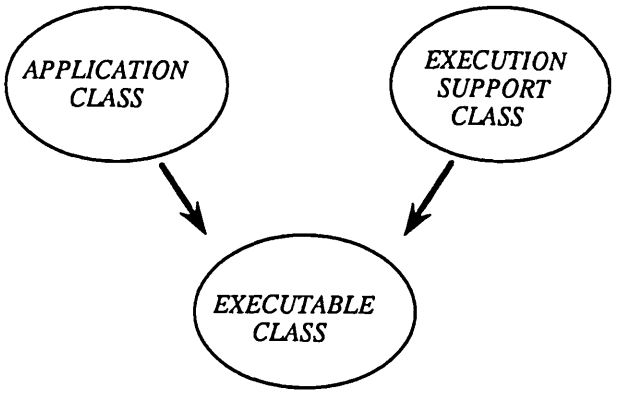


Figure 6.1: Generating Executable Classes

components from their lightweight counterparts [Bayan et al., 89], [Atkinson et al, 90]. An executable, or "heavyweight", version of a normal virtual node class is generated by defining a subclass which also inherits the properties of an appropriate *execution support class*, as illustrated in Figure 6.1. Such a subclass, termed an *executable class*, not only inherits all the sequential and behavioural properties of the virtual node class in the normal way, but is also empowered to execute on the particular type of machine associated with the executable class.

Suppose, for example, that a version of the class *VN_DATA_BASE* mentioned above was required to execute on a VAX under VMS. Such a component would be generated by defining a class which was an heir both of *VN_DATA_BASE* and a suitable execution support class :-

```
class VAX_EXECUTABLE_DATA_BASE                                    /6.3/
    inherits VN_DATA_BASE, VAX_VMS_EXECUTION_SUPPORT;
end;
```

where *VAX_VMS_EXECUTION_SUPPORT* is an execution support class with the general form :-

```
class VAX_VMS_EXECUTION_SUPPORT                                   /6.4/
    introduces LOAD_ONTO (PN : in VAX_VMS_PHYSICAL_NODE);
        :
            -- services provided by processes in the VAX/VMS environment
        :
end;
```

Like the behavioural classes of the previous chapter, execution support classes are not classes in the normal sense, since they cannot be instantiated independently.

Their only function is to serve as parents of virtual node classes to endow them with the quality of executability. Another property they share with behavioural classes is that they have no body. They cannot, in fact, be defined by applications programmers, but are supplied with a DRAGOON development environment as an interface to its (cross) compilation facilities. Executable classes differ from behavioural classes, however, in that they are not identified by a special keyword. Conceptually, executable classes are part of a separate hierarchy distinct from the application hierarchy originating at the class *APPLICATION_OBJECT* - the root of the "execution support hierarchy" being the class *EXECUTION_SUPPORT*.

Execution support classes are distinguishable from normal application classes by the inclusion of the method *LOAD_ONTO* in their interface. The role of this special method, which appears in the interface of every execution support class, will be described more fully in the next section. Since it is the main distinguishing feature of such classes this name is not available for use in normal application classes.

As well as this special *LOAD_ONTO* method, execution support classes may export methods representing services provided by processes in the corresponding environment. Suppose, for example, that the VMS operating system provided with VAX computers permitted processes under its control to be assigned an execution priority, denoted by an integer value. In the DRAGOON model this priority is a property of "heavyweight" executable objects which may be updated and obtained by external agents. To model this feature, the execution support class for processes in the VAX/VMS environment, *VAX_VMS_EXECUTION_SUPPORT*, would probably export the following additional methods :-

```
    :
procedure SET_PRIORITY (P : in POSITIVE);
function CURRENT_PRIORITY return POSITIVE;
    :
```

Submission of the class *VAX_EXECUTABLE_DATA_BASE* to the compilation system will give rise to a new class which extends the interface of the class *VN_DATA_BASE* by the three methods of the execution support class. However, the fact that this class inherits from an execution support class signals that it should not just be inserted into the application library like normal classes, but should be completely compiled and linked into an executable module for the appropriate machine - in this case a VAX under VMS. The class *VAX_VMS_EXECUTABLE_DATA_BASE*, therefore, does not just *describe* the interface of objects intended for execution in a VAX/VMS environment, it *is* an executable module (e.g. relocatable binary) which can be directly loaded onto a VAX and executed. In other words, the executable classes denote modules upon which all the translation steps necessary to make direct execution possible have been performed. This is not the case with normal classes, on

the other hand, which are merely textual templates describing the form of objects, but not actually defining objects which can be executed *independently.* Such classes represent "lightweight" objects which can only be instantiated within the thread of control of "heavyweight" objects.

This is a very important aspect of the DRAGOON approach to developing distributed systems. In the vast majority of development environments for commercial and industrial software, some form of translation must be performed on high-level code to turn it into a form that may be executed, and in the case of distributed systems this takes place between the programming and configuration phases of system building. The programming phase deals with the *description*, in terms of high level code, of the components to be distributed, while the configuration phase is concerned with actually distributing executable versions of the modules over the network. Few existing systems, however, attempt to handle the transition between these two forms in a uniform way. By capturing the conceptual step of translating a textual template into a directly executable module DRAGOON permits all aspects of the software development process to be handled within the object-oriented framework and by a single language. This, in turn, permits the many different executable versions of a component to be managed and stored just like all the components in the system.

6.6 Physical Node Objects

The previous sections have outlined how classes should be structured in order to model potentially distributable virtual node objects, and how these can be translated into truly distributable (i.e. executable) objects using execution support classes. In order to build a complete functioning distributed system, however, it is also necessary to have a way of representing the hardware architecture of the system, and the different types of processors which it contains.

There are many different types of machines and processors that could be used to form the nodes of a distributed system, ranging from the largest mainframes, to the smallest microprocessors. In the former case, the nodes encapsulate extensive hardware and "systems" software resources providing a wide range of facilities to user processes, while in the latter case there is likely to be very little native systems software, and far fewer facilities offered. From the point of view of the application processes, however, the distinction between which services of the machine they are running on is provided by the hardware or systems software is irrelevant. What is important is the the actual range of services provided by the machine and the type of processes which it can execute. As far as configuring distributed systems is

concerned, therefore, all the various nodes in the systems can simply be regarded as *abstract processing machines*, which make a set of services available to application processes, and are able to run suitable (i.e. correctly compiled) executable objects.

To remain faithful to the goal of modelling all aspects of system construction within a uniform object-oriented framework, in DRAGOON these abstract processing elements are represented as special *physical node* classes, which present the services offered by the machine as methods. Assuming that the communications systems provides full-logical interconnection between the nodes of a network, a particular network architecture corresponds to a set of instances of such physical node classes.

A physical node class representing the VAX/VMS abstract machine, for example, on which instances of the *VAX_EXECUTABLE_DATA_BASE* (6.3) class are intended to run, would have a specification of the form :-

```
class VAX_VMS_PHYSICAL_NODE                                        /6.5/
    introduces ACTIVATE (P : in VAX_VMS_EXECUTION_SUPPORT);
    :
        -- services provided by VAX/VMS machines
    :
end;
```

As this class illustrates, the structure of such classes is, not accidently, very similar to that of execution support classes. Like such classes physical node classes conceptually reside in another special class hierarchy rooted at the class *PHYSICAL_NODE*, and are distinguished by inclusion of the special method, *ACTIVATE*, in their interface. Moreover, like execution support classes they usually export an additional set of methods, which in the case of physical node classes represent the services provided by the abstract machine for use by the executable objects. These services correspond to what are often called "system calls" in normal programming parlance. A typical example is a call to the "system clock". Other examples include the opening of communication ports, or the invocation of device drivers.

Assuming that VAX/VMS machines permit application processes to ascertain the current time from the system clock, and perhaps determine the total number of currently active processes, methods of the following form would appear in the interface of *VAX_VMS_PHYSICAL_NODE*, in addition to *ACTIVATE* :-

```
    :
function CLOCK return TIME;
function ACTIVE_PROCESSES return NATURAL;
    :
```

6.7 Software and Hardware System Components

Using execution support and physical node classes of the form introduced above it is possible to represent all the different kinds of hardware and software components that need to be manipulated in the configuration phase to construct complete distributed systems. The processors in the target network are modelled by instances of physical node classes, while the software modules distributed amongst them for execution are represented by instances of executable classes.

Suppose, for example, that a distributed system was required in which three software modules were distributed over a network of two VAX/VMS machines. The two VAXes would be represented by instances of the class *VAX_VMS_PHYSICAL_NODE* (6.5), referenced by instance variables *VAX_A* and *VAX_B*, say :-

```
    :
VAX_A, VAX_B : VAX_VMS_PHYSICAL_NODE;
    :
```

while the software modules would be represented as instances of subclasses of *VAX_VMS_EXECUTION_SUPPORT* (6.3), identified by instance variables *EXE_1*, *EXE_2* and *EXE_3*, say. Except in the extremely unlikely circumstance in which all three of the modules were required to have precisely the same functionality these will be descendents of different application classes.

Although the entities modelled by executable classes are different in kind to those represented by normal classes, they are nevertheless *software* components which may be generated dynamically during the execution of the system. Like instances of normal classes, therefore, executable objects are generated by execution of the *CREATE* method inherited from the normal application classes. The entities modelled by physical node classes, on the other hand, are not software modules, but abstract processing machines largely, if not exclusively, composed of physical hardware. It is not possible therefore for applications software to generate new physical node objects by invoking *CREATE* operations as if they were software modules. Unlike the instantiation of an executable class, which in operating system terminology only involves the activation of a new process, the instantiation of a physical node class involves the creation of new hardware - something which is extremely difficult for a computer program to do ! It is possible to introduce new instances of physical node classes into a running distributed system, but this is driven by the availability of processors and networking hardware, rather than by the application software.

There is no predefined *CREATE* operation associated with physical node classes, therefore. Instead, the instance variables defining the initial configuration of the network are assumed to contain references to physical node objects on elaboration, rather than the usual "null" reference.

Software to Hardware Allocation

Having introduced the various different software and hardware components, and a means of identifying them, the final stage in system construction is to indicate how the software modules are to be distributed over the machines in the network. This is where the special methods *LOAD_ONTO* and *ACTIVATE* described above come into play.

Executable objects generated by invocation of the *CREATE* method are conceptually brought into existence on the machine on which the *CREATE* method is executed. Unless this is the required location for the object, it must then be moved to the appropriate site in the network by means of the *LOAD_ONTO* method exported in its interface. Suppose, for example, that the object generated by invocation of *CREATE* through *EXE_1* was generated on *VAX_A*, but was actually required to execute on *VAX_B*. The object would be moved to this machine by invoking its *LOAD_ONTO* method with *VAX_B* as argument :-

```
    :
EXE_1.LOAD_ONTO(VAX_B);
    :
```

This has the effect of transferring the object referenced by *EXE_1* to the machine *VAX_B*. Simply transferring the relocatable binary code[1] represented by the object into the memory of the target machine, however, does not make it a functioning component of the distributed system. To turn the *executable* object into an *executing* object capable of responding to method calls the machine on which it is intended to run must be made aware of its existence so that it can load it into main memory and begin its execution. This is achieved by means of the *ACTIVATE* method exported by physical node objects.

To activate the executable object *EXE_1* just transferred to *VAX_B*, for example, the *ACTIVATE* method of *VAX_B* has to be invoked with parameter *EXE_1* :-

```
    :
VAX_B.ACTIVATE(EXE_1);
    :
```

In the same way that the *START* method starts the *thread* of an active object, invocation of the *ACTIVATE* method start the execution, or *elaboration* in Ada terminology, of the whole *object* supplied as its argument. The *ACTIVATE* method is in a sense, therefore, the analogue of the *START* method for "heavyweight" processes. An important difference, however, is that the *ACTIVATE* method belongs to the machine

[1] and possibly initialisation data supplied as parameters to the *CREATE* operation

responsible for running the object, while the *START* method belongs to the active object itself.

Naturally, if an executable object is derived from an active object (i.e. its "application" parent is an active class) then after activation of the object by *ACTIVATE* it is also necessary to start the thread by the *START* method in the normal way.

Together with the *CREATE* and *START* methods of normal classes, the *LOAD_ONTO* and *ACTIVATE* methods of executable and physical node classes provide the basis for the overall execution model of DRAGOON software for both distributed and non-distributed systems. In general, to bring about the execution of an executable object referenced by the variable *EXE* on a physical node referenced by the variable *P_N* the following method calls must be issued :-

```
    :
EXE.CREATE;
EXE.LOAD_ONTO(P_N);
P_N.ACTIVATE(EXE);
EXE.START;
    :
```

Moreover, invocation of the methods must occur in this order. An attempt to execute any of the above methods before the preceding operation has been performed will generally result in the raising of an exception. It is not possible, for example, to "load" an object which has not been "created", to "activate" an object on a machine onto which it has not been "loaded", or to "start" the thread of an object which has not yet been "activated" (because it is not actually executing). There are two cases in which certain operations in this sequence need not be performed, however. If the object is passive it is clearly not necessary to invoke the *START* method (in fact, it is impossible because it does not possess such a method), and if an object is to be executed on the machine on which it was created it is not necessary to invoke the *LOAD_ONTO* method.

From the point of view of the execution model a non-distributed system is simply viewed as a trivial uni-node network composed of one executable object running on one physical node object. In principle all the operations outlined above still have to be performed, but clearly some of them are redundant in this case, and so may not have to be issued in practise. This depends on the user-interface offered by the execution environment.

The question of *who*, or *what*, actually performs these variable declarations and method invocations will be considered below in section 6.8. First it is necessary to examine in more detail the nature of the *ACTIVATE* and *LOAD_ONTO* methods.

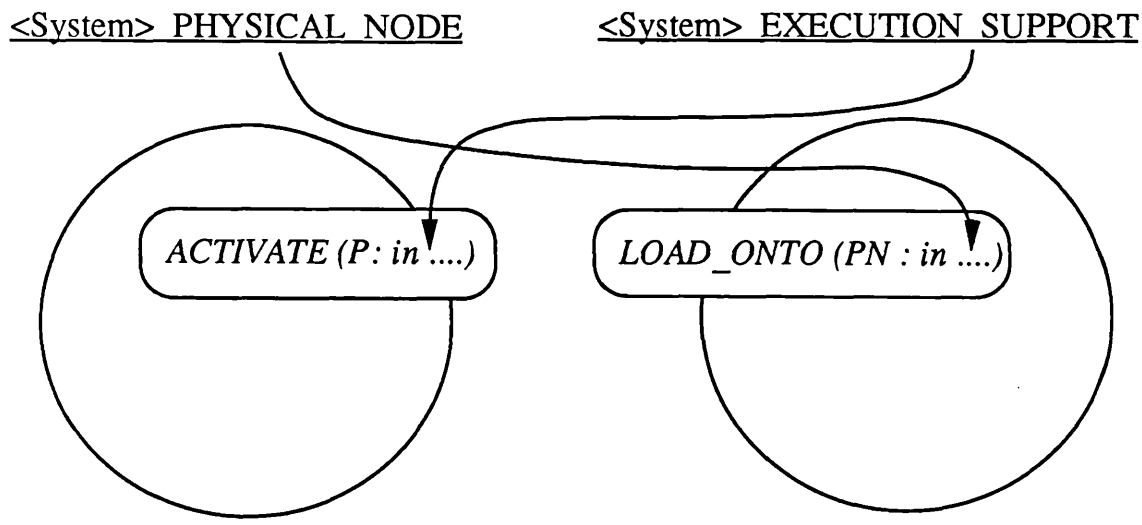


Figure 6.2: Canonical Pairs

6.7.1 Physical Node/Execution support Pairs

The *ACTIVATE* and *LOAD_ONTO* methods of physical node and executable objects perform complementary roles. The first provides a physical node object with "references" to the executable objects allocated to it for execution, while the second supplies an executable object with a reference to the physical node object on which it has been allocated for execution. When combined with the normal typing rules of the language, this technique provides an elegant means of modelling the different kinds of processors that may exist in heterogeneous networks, and the different kinds of software modules for execution on them.

One of the most basic principles of software development using compiler technology is that an executable module produced by a compilation system is targeted to one particular type of machine, and cannot be executed on another type with a different architecture or systems software. Conversely, a given computer can only execute software modules specifically targeted to that type of machine.

This fundamental principle is neatly captured in DRAGOON using executable and physical node classes by ensuring that for every physical node class there is a corresponding execution support class, and vice versa. This is the case with *VAX_VMS_EXECUTION_SUPPORT* (6.4) and *VAX_VMS_PHYSICAL_NODE* (6.5) classes introduced above. The former captures the qualities that an application class must inherit in order to be able to execute on a VAX/VMS machine while the latter defines the properties of the machines acting as processors for such objects.

For each different kind of processor architecture supported by the DRAGOON

(cross) compilation system there is at least one pair of associated execution support and physical node classes. Moreover, as illustrated figure 6.2, the parameter type of the physical node class's *ACTIVATE* method is the corresponding execution support class, while the parameter type of the execution support class's *LOAD_ONTO* method is the corresponding physical node class. The normal typing rules of DRAGOON, therefore, ensure the correct match of executable objects to physical node classes. In the case of a VAX/VMS system, for example, it is only possible to load descendents of *VAX_VMS_EXECUTION_SUPPORT* onto *VAX_VMS_PHYSICAL_NODE* objects and vice versa. In a heterogeneous network, therefore, in which there is a number of different types of processor, and hence executable objects, this prohibits inappropriate allocation of software modules to physical nodes.

The careful reader will notice that it is not actually possible to generate classes with this relationship in normal user-defined DRAGOON programs because of their mutual dependence. Not only are the classes clients of each other, but this dependency is expressed in their interfaces since each uses the other as a method parameter type. It is consequently impossible to compile either of these classes since they both rely on the prior compilation of the other.

Such mutual dependencies are not supported for normal DRAGOON classes. It is important to remember, however, that physical node and execution support classes are not normal application classes that may be defined by the user, but predefined classes supplied with a DRAGOON environment. Although they are known to the DRAGOON compilation system, therefore, and exist in the library of classes reachable by the applications programmer, they do not have to be "compiled" as such in the normal way. The role of these classes is in fact to provide an interface to the various possible system architectures supported by the environment and compilation system.

6.7.2 Device Drivers

The correspondence of physical node classes to execution support classes need not necessarily be a one-to-one relationship. In general, there may be many specialised versions of execution support and physical node classes for a basic machine architecture. In the case of execution support classes these would correspond to different kinds of processes supported in the same system, or of processes in specialised version of the system, while in the case of physical node classes, these correspond to different versions of the basic processor type extending or modifying the set of operations made available to the executable objects.

In DRAGOON these different specialised versions are defined as descendents of the most simple, or *canonical* versions of the physical node and execution support

classes. For every different type of machine architecture supported by the DRAGOON compilation system there must be at least a *canonical pair* which define the *LOAD_ONTO* and *ACTIVATE* methods. The classes illustrated in 6.2 are therefore a canonical pair. The specialised descendents of these classes inherit the *LOAD_ONTO* and *ACTIVATE* methods which they define, and so do not need to redefine them explicitly.

Subclasses of the canonical physical node classes are commonly used to model machines connected to particular types of devices. Suppose for example that a VAX in the network were connected to a laser printer. Although this VAX provides the same facilities as any other VAX in the network, and can only execute objects which have inherited the appropriate execution support, it also provides the additional set of operations associated with a laser printer. This particular machine in the network, therefore, is modelled by an instance of a subclass of *VAX_VMS_EXECUTION_SUPPORT* (the canonical physical node class) whose interface has been extended by the appropriate operations.

Another good example of where this facility would be needed is in the construction of the checkout points used in a distributed version of the central control system. It is unlikely that the machines used to provide processing power at the many checkout points in the store will be VAXes, but rather small, dedicated microprocessors. Moreover, not only are these microprocessors "attached" to the cash register device which they are responsible for controlling, but are completely "embedded" within them. The normal facilities provided by the embedded microprocessor type, therefore, are extended by a large number of operations used to control the cash register hardware.

A typical example of such an operation is the opening of the till storing the money. A common feature of many cash registers is that the till cannot be opened until the total bill of a customer is calculated, at which point it is opened automatically. To handle this interaction between software and hardware in DRAGOON it is necessary to model the *OPEN_TILL* operation as a method of the physical node class for cash registers.

Suppose for the sake of argument that the cash registers chosen for a particular version of the supermarket system were based on Motorola 68000 microprocessors with a canonical physical node class of the form :-

```
class MC_68000_PHYSICAL_NODE;                                        /6.6/
    introduces
        procedure ACTIVATE (P : in MC_68000_ EXECUTION_SUPPORT);
            :
                -- facilities provided by MC68000 processors
            :
```

```
end MC_68000_PHYSICAL_NODE;
```

where *MC_68000_EXECUTION_SUPPORT* is the corresponding canonical execution support class. The physical node class describing cash registers, in which the operations provided by such processors are extended by the facilities of the encapsulating device, would thus be modelled as a subclass of *MC_68000_PHYSICAL_NODE*. This extends its interface with the additional methods necessary for controlling the register hardware, such as *OPEN_TILL*.

```
class CASH_REGISTER_PHYSICAL_NODE                                    /6.7/
    inherits MC_68000_PHYSICAL_NODE;
    introduces OPEN_TILL;
        :
            -- other operations provided by the cash register hardware
        :
end CASH_REGISTER_PHYSICAL_NODE
```

Objects executing on an instance of this class can control the associated device directly by invoking the additional methods exported in the interface. Consider the checkout point objects, for example. As described in the previous chapter, while they are calculating customer bills such objects interact intensely with the data base in order to establish the price of the products selected. Once the total bill has been calculated, however, the checkout point objects need to arrange for the till of the cash register they are controlling to be opened ready for payment. To do this they need to be a *client* of the *CASH_REGISTER* object on which they are intended to run. The instruction to open the till can thus be issued simply by invoking the corresponding method in the normal way :-

```
    :
CR : CASH_REGISTER;
    :
CR.OPEN_TILL;                    -- activates till-opening action in the hardware
    :
```

There is one final problem, however, that must be addressed for this technique of issuing "system calls" to the underlying processing system to work, which is arranging for the instance variables of the physical node classes to refer to the appropriate machine. As mentioned above, there is no *CREATE* "method" associated with instances of physical node classes, since such instances do not correspond to software modules which can be generated dynamically. How then can instance variables such as *CR* be made to refer to the physical node to which the encapsulating software has been allocated for execution ?

The key point to remember is that the only mechanism for executing software in DRAGOON, even in a non-distributed system, is by inheritance of an appropriate

execution support class, and allocation of instances of the resulting subclass to a corresponding physical node object for execution. Moreover, this involves invocation of the *LOAD_ONTO* method inherited from the execution support class, with the reference of the machine on which the object is to execute supplied as an actual parameter. This is precisely the reference that needs to be assigned to instance variables such as *CR* above in order to provide executable objects with capability of invoking the system methods of the host machine.

The problem of assigning the appropriate reference to "physical node" instance variables is not a concern of the application class, therefore, but rather of the executable class generated from it by inheritance of an execution support class. When executable objects are required, such as *CHECKOUT_POINT* which need to invoke methods of the underlying system, the *LOAD_ONTO* method of the corresponding class must be redefined to assign its parameter to the necessary instance variables inherited from the application class. The executable class *MC_68000_CHECKOUT_POINT*, for example, which defines the executable classes needed to control the Motorola 68000 based cash registers, would be be of the form :-

```
class MC_68000_EXECUTABLE_CHECKOUT_POINT;                    /6.8/
    inherits MC_68000_EXECUTION_SUPPORT, CHECKOUT_POINT;
    redefines LOAD_ONTO;
end;

class body MC_68000_EXECUTABLE_CHECKOUT_POINT is
    procedure LOAD_ONTO (P : in MC_68000_PHYSICAL_NODE) is
    begin
        CR := P;
    end;
end;
```

This conforms to all the normal class redefinition rules except in so far as the method *LOAD_ONTO* is a special method which has a far greater effect than that described in the new body. As with the "redefinition" of *CREATE*, therefore, the redefinition of *LOAD_ONTO* is best thought of as defining *extra* operations performed when the method is invoked rather than providing an *alternative* redefinition of its entire functionality. The new body is able to assign the reference provided as parameter *P* to *CR*, and indeed to any other "physical node" instance variable in the body of the application class, since by the normal rules of inheritance these are visible in the body of subclasses.

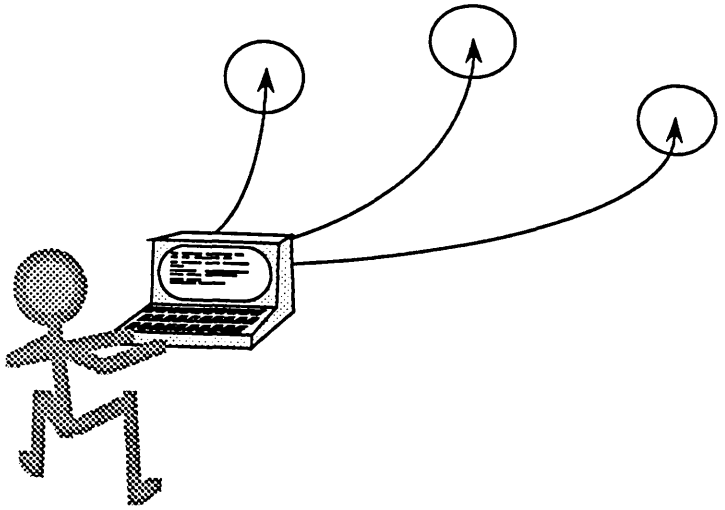


Figure 6.3: Interactive Configuration from Terminal Object

6.8 DRAGOON Execution Model

The previous section has described how the different kinds of components in distributed systems are modelled using physical node and execution support classes, and the various facilities available for configuring them. However, the question as to how these facilities are employed, and who or what is actually responsible for configuring systems, was left open.

At first sight there would appear to be two basic strategies by which the objects needed to produce a functioning system in DRAGOON could be generated and configured - *interactively*, under the control of a human operator, and *automatically* under the control of some form of configuration program.

Interactively. Pursing the object-oriented model to its limit, the terminal (or shell in UNIX terminology) which the human operator uses to perform the configuration actions can be regarded as an *active* object which permits the normal declarations and operations possible in a DRAGOON class to be performed interactively, as illustrated in figure 6.3. The operator at this *terminal object* may therefore define instance variables (c.f. UNIX shell variables) and use them to invoke methods (c.f. command line instructions) in order to arrange for the execution of executable objects in the manner described above.

This is true for non-distributed as well as distributed systems. To execute the non-distributed version of the supermarket control system represented by the class *SUPERMARKET* (5.11), for example, it would first be necessary to define (by in-

structions at the command line) an instance variable of the appropriate executable subclass of *SUPERMARKET*, and then generate an executable object by invocation of the *CREATE* method. Since there is only one machine in the network, and the executable object was generated on the machine on which it is to run, it is not necessary to invoke the *LOAD_ONTO* method of the object in this case. It is still necessary to invoke the *ACTIVATE* method of the host machine, however, in order to load the corresponding binary module and begin its execution. A terminal object, therefore, needs some pseudo-variable, *HOST* say, which contains a reference to the physical node object to which the terminal is connected. The executable "supermarket" would therefore be activated by invoking the *ACTIVATE* method through the pseudo-variable *HOST*. Finally, since *SUPERMARKET* is an active class, the thread of the executable object just activated needs to be started by invoking the *START* method.

The construction of distributed systems is just a generalisation of this scenario, in which physical objects and numerous executable objects are generated by instructions issued from the *terminal object*, and the *LOAD_ONTO*, *ACTIVATE* and *START* methods are used to define the allocation of software modules to physical nodes, and the communication pattern of the executable objects.

This view of system generation and execution is an idealistic model in which the object-oriented framework of DRAGOON is pushed to its limit. To what extent this is actually realised in practice is more a concern for the environment/operating system in which the software is executed than for the language, itself. It is certainly possible to provide a fully object-oriented environment for DRAGOON, perhaps similar to that of Smalltalk, where sophisticated mouse driven browsers are provided to analyse the contents of the library and invoke the methods of executable objects. On the other hand, the system may provide a very primitive user interface not supporting the full object scheme for activating executable objects.

Automatically. Since all the system configuring operations are performed by normal mechanisms of the object-oriented paradigm, another natural approach is to define the required configuration steps in a "class", rather than to enter them interactively from a terminal object. The "execution" of an instance of such a "class" would therefore perform the necessary configuration steps automatically. Such an instance is, therefore, closely analogous to the *configuration programs* of languages such as CONIC, which are used to build and configure distributed systems in the second phase of program construction.

Two immediate questions spring to mind with this approach, however - who is responsible for instantiating this class, and how is an instance executed once produced ? Because such a class manipulates "heavyweight" executable objects and

physical node objects, and therefore essentially encapsulates the whole network, it would at first sight seem to be very different from the normal application classes of DRAGOON. The operations performed by instances of such classes, such as the invocation of *LOAD_ONTO* and *ACTIVATE* methods cannot be compiled directly into binary code as normal, but must be implemented by appropriate calls to the services of the underlying operating system. Moreover, the references to objects are not implemented as memory addresses, but as network addresses for the machines and processes concerned.

Despite the fact that operations performed by such a class are mapped into operating system commands rather than into machine code as with normal classes, instances of the class clearly still need to be executed on a machine in the network, typically the host. There is no reason, therefore, why the same execution model cannot also be adopted for such "configuration" classes - namely, that executable subclasses have to be generated by inheritances of the appropriate run-time support. This also means that the generation and activation of such classes can be performed for the terminal object in the same way as for normal application classes described above. The only difference is that whereas in the interactive case *all* configuration concerns have to be handled explicitly from the command line, in this case the activation of a single object can lead to the generation of a complete distributed system.

Integrated Execution Model

The two alternative approaches described above represent the two extremes. In the first, all configuration operations are performed interactively from an (active) terminal object by the user, while in the second, all the configuration operations are performed by an instance of a special class which exclusively manipulates physical node and executable ("heavyweight") processes. It is not necessary to have such a sharp dichotomy, however.

The "automatic" approach described above, in which special "configuration" classes are used to perform the configuration operations necessary for constructing distributed systems, would on the face of it appear to divide DRAGOON *application* classes into two distinct sets - the normal application classes which are clients of other normal ("lightweight") classes, and the special "configuration" classes, which are clients of executable ("heavyweight") and physical node classes. However, because DRAGOON enables all the different types of object that may exist in a system to be distinguished by their ancestry, and the same execution model is used for both categories, there is no reason why a class cannot at the same time be a client of all these different kinds of classes. When a class is analysed by the compilation

system it can determine unambiguously whether a server object is "lightweight", in which case operations applied to them must be compiled into binary code as usual, or whether they are executable or physical node class in which case the operations must be mapped into the appropriate calls to the operating system.

The important point is that the compiler is able to determine purely from the class text what kind of objects are being manipulated, and how particular operations should be translated. It is not necessary, therefore, to have different categories of class according to the types of objects they manipulate, and in which phase of system construction they are used. All classes can manipulate all the three different kinds of object recognised in DRAGOON systems without ambiguity. The only difference between objects which are clients of one or more physical node or executable objects, and those which are not, is that they can not be translated into pure binary modules, but rely on calls to the underlying operating system to put some of their actions into effect.

This is the approach adopted in DRAGOON. There is no distinction between "configuration" and "normal" objects, or between the interactive and automatic configuration. Any DRAGOON class can be a client of any other kind of class without restriction, and can, in particular, therefore perform any of the actions required to configure a system. Nevertheless, even such a class can only be executed by generating the appropriate executable subclass, and instantiating it from the terminal object. It is entirely up to the programmer, therefore, how much of the configuration is performed "automatically" by objects in the system, or "interactively" from the terminal object.

6.8.1 Centralised Reconfiguration

This integrated model for executing objects and configuring systems has a number of important advantages as far as supporting dynamic reconfiguration is concerned. If a single object is used to construct a system in the style of the "automatic" approach it can play a role in monitoring and upgrading the configuration of the network, because it is also a normal executing object in the system just like an instance of the class *SUPERMARKET* (5.11) defining a non-distributed version of the system.

A class defining a distributed version of the supermarket control system would have a similar form :-

```
class DISTRIBUTED_SUPERMARKET;                                    /6.9/

with SUN_EXECUTABLE_CENTRALLER_CONTROLLER;
```

```
with VAX_EXECUTABLE_DATA_BASE;
with SUN_EXECUTABLE_FORWARDER;
with MC_68000_EXECUTABLE_CHECKOUT_POINTS;
with VAX_VMS_PHYSICAL_NODE, SUN_PHYSICAL_NODE;
with CASH_REGISTER_PHSYICAL_NODE;
with CENTRAL_CONTROLLER, CHECKOUT_POINT, DATA_BASE;
class body DISTRIBUTED_SUPERMARKET is

    CC : SUN_EXECUTABLE_CENTRAL_CONTROLLER;        -- declare variables for
    DB : VAX_EXECUTABLE_DATA_BASE;                 -- executable objects
    F : SUN_EXECUTABLE_FORWARDER;
    CP : array (1 .. 10) of MC_68000_EXECUTABLE_CHECKOUT_POINTS;

    VAX : VAX_VMS_PHYSICAL_NODE;          -- variables for physical node objects
    SUN : SUN_PHYSICAL_NODE;              -- assumed to contain references
    REGISTER : array (1 .. 10) of CASH_REGISTER;  -- immediately upon elaboration

thread

    CC.CREATE;                                                  -- create objects
    DB.CREATE;
    F.CREATE;
    for I in 1 .. 10 loop
        CP(I).CREATE;
    end loop;

    CC.LOAD_ONTO (SUN);      -- load executable objects onto appropriate machines
    DB.LOAD_ONTO (VAX);
    F.LOAD_ONTO (SUN);
    for I in 1 .. 10 loop
        CP(I).LOAD_ONTO (REGISTER(I));
    end loop;

    CC.SHARE(F);
    F.SHARE(DB);                          -- setup required system configuration
    for I in 1 .. 10 loop
        CP(I).SHARE(DB);
    end loop;

    SUN.ACTIVATE (CC);                          -- activate executable objects
    VAX.ACTIVATE (DB);
    SUN.ACTIVATE (F);
    for I in 1 .. 10 loop
        REGISTER(I).ACTIVATE (CP(I));
    end loop;

    CC.START;                                                   -- start objects
    F.START;
    for I in 1 .. 10 loop
        CP(I).START;
    end loop;

end DISTRIBUTED_SUPERMARKET;
```

An instance of this class activated from the terminal object will issue the nec-

essary operating system instructions to generate a distributed version of the supermarket system in which the central controller and forwarder objects execute on the same Sun workstation, the data base on a VAX, and the checkout points on MC6800 based cash registers.

Notice how similar this class is to the class *SUPERMARKET* defining a non-distributed version. Precisely the same method calls are performed to set up the logical configuration of the system (i.e. the population of application objects and their communication pattern) in both cases. The only difference is that the distributed version must also handle the physical configuration of the system (i.e. the number and type of physical node objects) and the allocation of executable objects to physical nodes.

As in the non-distributed version of the system, the thread of the configuring class *DISTRIBUTED_SUPERMARKET* need not terminate after it has set up the system, but by containing a loop may become a permanent active object which plays a role in the execution of the system. For example, it could provide a similar reconfiguration service in connection with forwarder objects as the non-distributed version, described in the previous chapter, by generating a new forwarder when the old one becomes full.

Such a reconfiguration is termed a *logical* or *software* reconfiguration since it changes only the logical structure of the application software. In distributed systems, in which the hardware in the network, or arrangement of software upon it, may also be change dynamically, other forms of the reconfiguration are possible. Should extra physical nodes become available during the execution of the program the configuring class could be programmed to move the existing objects onto the new processing sites, or to add new executable objects to the system to execute upon them. Such adjustment is termed *physical* reconfiguration since it effects the structure of the hardware, as well as, or instead of the software. An important feature of the DRAGOON approach is that all these different possibilities are programmed using the normal mechanisms of object-oriented programming. In contrast with systems such as CONIC, therefore, which use a limited form of "configuration language" to describe the final structure of distributed systems, this means that in DRAGOON the usual "flow-control" mechanisms (i.e. loops, conditional statements etc.) may be used in the configuration process.

However, the problem faced in the non-distributed version of the system, and left open in connection with the forwarder reconfiguration, also exists here - how does the *reconfiguring* object know when the reconfiguration should be performed ?

Environment Driven reconfiguration

One strategy is to include a method in the interface of the configuring object which the human operator may call from the terminal object when a reconfiguration is required. This method could either perform the reconfiguration itself, or "unblock" the thread to do the job. In the case of the "forwarder reconfiguration", for example, the configuring class (*SUPERMARKET* or *DISTRIBUTED_SUPERMARKET* in the non-distributed and distributed versions respectively) would export an additional method, *GENERATE_NEW_FORWARDER* say, which could be called from the terminal object to indicate that a new forwarder should be generated.

This approach is fine for reconfigurations in which the human user is genuinely the agent which detects or decides that the change is needed, such as those generally required as a result of a change in the encapsulating environment. The prime example are hardware reconfigurations where processors are added or removed from the system. As pointed out above, it is impossible for the applications software to issue commands which "generate" new hardware, so it must ultimately be notified of such changes by the external environment.

Since such reconfigurations are driven by changes in the environment, one of the major problems in this approach is ensuring that the objects affected by the change are in an appropriate state for the change to be made safely. In general, this requires the configuring object to ensure that the objects are in a *quiesent state* [Kramer and Magee, 88] by calling a number of extra methods specially defined for this purpose in the interface of application objects. Essentially this involves the configurer requesting permission to perform a change, and then subsequently informing the object of its completion.

System Driven reconfiguration

There are many reconfigurations, however, such as the "forwarder reconfiguration" under consideration, which depend on the state of the application software, rather than events in the external environment. Quite apart from the question of how the human operator can actually detect when such a state is reached, it is clearly more desirable for such reconfigurations to be performed automatically, than for the user to act as an interface between the application modules and reconfiguring agent.

The basic virtual node approach to distribution, in which the distributable objects developed in the programming phase are made as independent as possible from any configuration concerns, does not readily support automatic reconfiguration of this form. The decoupling of the programming and configuration concerns which provides much of the flexibility for building different system configurations, makes

it much more to arrange for interaction between the application objects and configuring object(s).

Since it is the application objects in the system which detect *when* a reconfiguration is needed (e.g. the central controller becomes aware when the forwarder is full) but the encapsulating object knows *how* a reconfiguration can be performed, the problem is to facilitate communication between them. In other words, how does the central controller object *notify* the encapsulating, system-configuring object that the forwarder is full and a new one is required ?

This cannot easily be achieved in many languages pursuing the virtual node approach, but is supported in a fairly straightforward way in DRAGOON because it is possible to arrange for an object and its creator to be mutual clients. In the case of the distributed supermarket system, for example, since the *CENTRAL_CONTROLLER* class is compiled first, the *DISTRIBUTED_SUPERMARKET* class responsible for configuring the system can be a client of it in the normal way. Once the *specification* of *DISTRIBUTED_SUPERMARKET* has been compiled, however, it is possible to define a subclass of *CENTRAL_CONTROLLER* which is a client of *DISTRIBUTED_SUPERMARKET*. By judiciously ordering the compilation of class specifications and bodies, therefore, the polymorphism properties of DRAGOON can be used to arrange for the (executable) application objects and the system-configuring object to be mutual clients. Communication between them can thus be programmed using method calls in the normal style.

6.8.2 De-centralised Reconfiguration

The above discussed has demonstrated how "centralised" configuration schemes, similar to those of CONIC, can be handled in DRAGOON, with full support for both user driven and system driven reconfiguration. However, because of the fully integrated execution model, it is possible to distribute the responsibility for system configuration amongst executable objects at sites throughout the system.

It is possible, for example, to define an application class, *FORWARD_MONITOR*, say, whose sole function is to generate a new *FORWARDER* object when requested to by the central controller :-

```
class FORWARDER_MONITOR is
    procedure CREATION_SITE (S : in SUN_PHYSICAL_NODE);
    procedure GENERATE_NEW_FORWARDER
        :
```

An alternative strategy is to allow the central controller to perform the reconfiguration itself, since it can be a client of both physical node and executable classes.

Chapter 7

Translation into Ada

One of the main design goals of DRAGOON was that it should be readily translatable into Ada for execution. This chapter describes how, despite the *prima facie* incompatibility of the languages, DRAGOON is translated into Ada in a fairly succinct and natural style. The aim is not to provide an exhaustive and detailed definition of all aspects of the transformation, but rather to concentrate on the way in which the principal features of DRAGOON, particularly those most alien to Ada, are handled.

7.1 Possible Implementation Strategies

Thanks to its extensive modularity, concurrency and typing mechanisms Ada provides more opportunities for supporting object-oriented programming than most other conventional imperative languages, such as Pascal and Modula2. In fact, to a limited extent it is possible to emulate all the fundamental features of the object-oriented paradigm in Ada, including inheritance, polymorphism and dynamic binding [Seidewitz, 87]. For reasons explained below, however, most of these *direct* simulation techniques can not be used in the implementation of a general object-oriented language like DRAGOON. Not only do they fail to support the object-oriented mechanisms in their full generality, but tend to be disjointed, and impossible to use together. Nevertheless, before discussing the particular implementation strategy developed for DRAGOON it is helpful to review these different possibilities.

7.1.1 "Abstract State Machine" Packages

There are, in fact, three basic techniques for representing objects and classes in Ada. The most direct representation of an object is as a state-encapsulating package (sometimes termed an abstract state machine) exporting a set of subprograms (c.f. methods) which can be used to access and update the encapsulated state. A product

descriptor object, for example, of the kind used in the supermarket control system, and defined in DRAGOON as an instance of the class *PRODUCT_DESCRIPTOR* (3.4), could be represented explicitly in Ada as an "abstract state machine" package of the form[1] :-

```
with PRODUCT;                                                    /7.1/
package ASM_PRODUCT_DESCRIPTOR is
    function PRICE return PRODUCT.VALUE;
    procedure SET_PRICE_TO (PRICE : in PRODUCT.VALUE);
    function BAR_CODE return PRODUCT.CODE_NUMBER;
        :
            -- similarly for other "methods" of PRODUCT_DESCRIPTOR
        :
end ASM_PRODUCT_DESCRIPTOR;


package body ASM_PRODUCT_DESCRIPTOR is
    INTERNAL_BAR_CODE : PRODUCT.NUMBER;
    INTERNAL_PRICE : PRODUCT.VALUE;
        :
            -- other attributes, and subprogram bodies
        :
end ASM_PRODUCT_DESCRIPTOR;
```

Superficially, this package is very similar to the class, *PRODUCT_DESCRIPTOR* (4.3), but represents a product descriptor *object*, rather than a *template*.

Using generic packages, this approach can be extended to achieve something of the notion of classes. A generic state-encapsulating package with this interface defines an object template from which multiple, structurally identical instances can be generated. However, generic packages are static entities which can only be instantiated at compile time, and thus do not support the concept of dynamically instantiatable objects identified by references.

This approach is probably the least satisfactory of the three, therefore, because it does not provide a faithful implementation of classes as dynamically instantiatable objects types, let alone support for the accompanying mechanisms of inheritance, polymorphism and dynamic binding.

7.1.2 Tasks

The second technique is to represent objects as tasks. Since Ada supports the notion of task types as well as just instances, this approach has the immediate advantage

[1]To distinguish between the languages the keywords of Ada code appear in **bold**, while those of DRAGOON appear in the usual **sans serif**.

of supporting dynamically instantiatable classes and reference semantics. In this approach the class *PRODUCT_DESCRIPTOR* (3.4) would be represented by a task type of the form :-

```
task type PRODUCT_DESCRIPTOR_TASK is                                    /7.2/
    entry PRICE (P : out PRODUCT.VALUE);
    entry SET_PRICE_TO (PRICE : in PRODUCT.VALUE);
    entry BAR_CODE (B : out PRODUCT.CODE_NUMBER);
        :
            -- similarly for other "methods" of PRODUCT_DESCRIPTOR
        :
end PRODUCT_DESCRIPTOR_TASK;
```

Notice that methods such as *PRICE* and *BAR_CODE* defined as functions in the DRAGOON representation of such objects have to be defined as value returning entries in this representation.

One of the main advantages of this approach is that it supports the notion of concurrency, and enables objects to be active. It can also be used to realise a form of dynamic binding because Ada permits a task type to define several different "accept" statements (c.f. method bodies) for each entry (c.f. method) exported in the interface. The body of the task *PRODUCT_DESCRIPTOR*, for example, could contain several different accept statements for the entry *PRICE*. By parameterising the task with a flag indicating which of the accept statement is to be executed, it is possible for different instances of a single task type to provide alternative implementations for the entry concerned. Moreover, since such instances are identified by access values of the same type, and may all be assigned to a single variable of that type, the particular implementation of the entry executed in response to an entry call is established (bound to the entry name) dynamically.

As with the first approach, above, the main problem with using task types for implementing classes is that they do not provide any support for inheritance. There is also the additional problem that tasks, unlike classes, can not be library units.

7.1.3 Abstract Data Types

The final strategy for representing objects in Ada is based on the notion of abstract data types. Instead of modelling the object abstraction as a package representing an abstract state machine, the abstraction is defined by a package defining an abstract data type. It is not the package exporting the type and associated subprograms which itself represents the objects, but variables storing values of the data type. The package defining the abstract data type corresponds more to a class, therefore, than to an object.

In this approach the class *PRODUCT_DESCRIPTOR* (3.4) would be represented as an abstract data type of the form

```
with PRODUCT;                                                    /7.3/
package ADT_PRODUCT_DESCRIPTOR is

    type OBJECT is private;

    function PRICE (OB : in OBJECT) return PRODUCT.VALUE;
    procedure SET_PRICE_TO (OB : in OBJECT; PRICE : in PRODUCT.VALUE);
    function BAR_CODE(OB : in OBJECT) return PRODUCT.CODE_NUMBER;
        :
            -- similarly for other "methods" of PRODUCT_DESCRIPTOR
        :
private

    type OBJECT is record
        INTERNAL_BAR_CODE : PRODUCT.CODE_NUMBER;
        INTERNAL_PRICE : PRODUCT.VALUE;
        INTERNAL_STOCK_CODE : PRODUCT.CODE_NUMBER;
        INTERNAL_LABEL : PRODUCT.LABEL;
    end record;

end;
```

Product descriptor objects are generated by elaboration of variables of type *ADT_PRODUCT_DESCRIPTOR.OBJECT*, and operations on the objects are performed by supplying the appropriate variable as a parameter of the corresponding subprogram. Each of the methods in the class is, therefore, mapped into a subprogram with an extra parameter for receiving the "state" of the object upon which the operation is to be performed. The type *OBJECT* is defined as "private" so that clients of the class can not manipulate the state of instances directly, but only by means of the subprograms defined in the interface.

One problem with this simple mapping into an abstract data type is that it fails to provide the reference semantics, and the associated facility for dynamically generating objects, which is such a fundamental aspect of object-oriented languages. This is easily overcome, however, by making the exported type an access type rather than a static type :-

```
with PRODUCT;                                                    /7.4/
package ADT_PRODUCT_DESCRIPTOR is

    type STATE is private;
    type OBJECT is access STATE;

    function PRICE (OB : in OBJECT) return PRODUCT.VALUE;
        :
            -- subprogram specifications as above
        :
```

```
private
    :
end;
```

This models the references and object creation semantics of DRAGOON much more faithfully, since variables of type *ADT_PRODUCT_DESCRIPTOR.OBJECT* are now access variables to which different references may be assigned dynamically, and product descriptor objects are generated dynamically by means of the "new" allocator.

Inheritance

The main advantage of this approach is that it provides limited support for two of the other important mechanisms associated with classes/objects, inheritance and polymorphism. Something of the effect of inheritance is provide by the "subtype" and "derived type" mechanisms of Ada. If, for example, a new type, *PERISHABLE_OBJECT* were derived from the type *OBJECT* defined in (7.4) above by a declaration of the form :-

type *PERISHABLE_OBJECT* **is new** *PRODUCT_DESCRIPTOR.OBJECT*

then all the subprograms associated with *OBJECT* in the package *PRODUCT_DESCRIPTOR* are also applicable to values of the derived type *PERISHABLE_OBJECT*. In other words, *PERISHABLE_OBJECT* "inherits" the subprograms associated with the type *OBJECT* from which it was derived. Moreover, it is possible to extend the set of "methods" applicable to the derived type *PERISHABLE_OBJECT* by defining new methods taking parameters of this type.

The derived type mechanism of Ada, therefore, provides a faithful representation of inheritance with respect to the *operations* associated with a "class". In particular, it is possible to specialise a "class" by expanding the set of "methods" applicable to it. The subtyping mechanism - involving the specification of dynamic constraints on the set of value that entities of the subtype may assume - provides similar possibilities. While derived types are incompatible, however, values of a subtype may be assigned to variables of the parent type.

The main shortcoming of both the subtyping and derived type approaches is that they do not permit the set of state variables associated with an abstraction to be extended. Neither of these mechanisms, therefore, are able to directly support the specialisation of a product descriptor template to a perishable descriptor template, because it is not possible to introduce the additional attribute of type *SIMPLE_CALENDAR.DATE*.

[Perez, 88] describes how this problem can be ameliorated to some extent by embedded "inherited" state variable in a record containing the additional state variables

introduced by a subclass. The resulting type, however, is still not type compatible with the parent type.

Polymorphism

The "abstract data type" approach also supports a limited form of polymorphism by means of the variant record facility. As in Pascal and Modula 2, variant records permit data structures with different numbers and types of fields to be treated as if they were of the same type. The different kinds of records are actually regarded as different "variants" of a single variant record type controlled by a *discriminant.* Using the type *PRODUCT_TYPE* (3.1) defined in chapter 3 as a discriminant, the different kinds of product descriptors used in the supermarket control system (modelled in DRAGOON by the classes *PRODUCT_DESCRIPTOR* (3.4), *PERISHABLE_DESCRIPTOR* (3.7) and *ELECTRICAL_DESCRIPTOR* (3.8)) would be represented as variants of a record of the form :-

```
with PRODUCT, SIMPLE_CALENDAR, FUSE;                                    /7.5/
package VARIANT_ADT_DESCRIPTOR is
    type STATE(FORM : in PRODUCT_TYPE) is private;    -- a parameterised type
    type OBJECT is access STATE;
    function PRICE (OB : in OBJECT) return PRODUCT.VALUE;
        :
            -- subprogram specifications as in (7.3) above
        :
private
    type STATE(FORM : in PRODUCT_TYPE) is record
        INTERNAL_BAR_CODE : PRODUCT.CODE_NUMBER;
        INTERNAL_PRICE : PRODUCT.VALUE;                  -- common attributes
        INTERNAL_STOCK_CODE : PRODUCT.CODE_NUMBER;
        case FORM is
            when PERISHABLE =>
                INTERNAL_DATA : SIMPLE_CALENDAR.DATE;
            when ELECTRICAL =>
                INTERNAL_RATING : FUSE.RATING;
            when BASIC =>
                null;
        end case ;
    end record;
end;
```

Using this package, instances of the classes *PRODUCT_DESCRIPTOR*, *PERISHABLE_DESCRIPTOR* and *ELECTRICAL_DESCRIPTOR* would be regarded as *BASIC*, *PERISHABLE* and *ELECTRICAL* variants of the record type *VARIANT_DESCRIPTOR.STATE*

respectively. Since they are thus of the same Ada type they can be assigned to access variables of type *VARIANT_DESCRIPTOR.OBJECT*. This approach, therefore, supports the desired polymorphic behaviour with respect to the different kinds of descriptors in the supermarket system.

7.1.4 DRAGOON Translation Strategy

Using the different techniques outlined above, it is possible to achieve something of the effect of all the principal object-oriented mechanisms outlined in the previous chapters. Although it does not provide full support for inheritance, therefore, Ada probably has more in-built features supporting "object-oriented programming" than any other language not generally credited with this title.

The level of support provided by Ada is quite adequate for implementing object-based *designs* which do not rely on the full power of inheritance, polymorphism or dynamic binding, and is largely responsible for the current popularity of such methods in connection with Ada. In implementing designs constructed using such methods the most appropriate mapping can be used for each particular circumstance. Booch [87], for example, uses all three different techniques for modelling classes and objects depending on the particular context in which they appear.

As they stand, however, none of these translation strategies is acceptable for implementing a general object-oriented language like DRAGOON because they each support only a certain subset of the required properties. One of the most important aspects of DRAGOON is its use of the "object" as a *unifying* concept acting as a focus for all the mechanisms supporting reuse, concurrency, distribution and reconfiguration. It is important, therefore, that its translation into Ada permits *all* these to be usable together in the construction and manipulation of classes and objects. The main problem in using Ada as the implementation language for DRAGOON, therefore, is the basic orthogonality of its features simulating inheritance, polymorphism and dynamic binding.

If a class is modelled by a task type, for example, in order to take advantage of the dynamic binding and reference semantics this offers, it is not possible to use inheritance. If derived types are used, on the other hand, to make use of inheritance, it is not possible to use dynamic binding. These two mechanisms, therefore, are mutually exclusive in a direct implementation of objects in Ada.

Alternatively, consider the three different versions of product descriptor objects. As demonstrated above, it is possible to handle the polymorphism resulting from their type compatibility using variant records. A client of *PRODUCT_DESCRIPTOR*, such as a queue of such objects, can manipulate all the different versions of the objects, since they are all represented by the same Ada type.

The major drawback of this technique, however, is that all these different versions need to be known and included in the record *at the time of definition.* It is not possible to introduce new subclasses of *PRODUCT_DESCRIPTOR* into the system, and make them type compatible with access variables corresponding to existing classes, without redefining the variant record. This has serious repercussions throughout the system because the Ada code for all the clients of the class depends upon it. This approach for achieving polymorphism is completely incompatible with inheritance, therefore, since it involves the recompilation of all the ancestors and clients of a class when a new subclass is defined.

Apart from the limited possibilities provided by variant records polymorphism is in direct conflict with the strong typing principles of Ada. If variant records are not used to representing objects, therefore, the translation needs to break the strong typing rules of Ada using the *UNCHECKED_CONVERSION* facility. In view of these problems, therefore, and the unsuitability of the translation strategies outlined above, the translation adopted for DRAGOON aims to support inheritance, polymorphism and dynamic binding in a more uniform way, as described below.

7.2 Representation of Object State

Of the three strategies for implementing classes and objects outlined in the preceding section, the translation of DRAGOON into Ada has most in common with the third "abstract data type" approach. In particular, it adopts the same basic philosophy of representing classes as "abstract data types" (i.e. types with associated subprograms), objects as dynamically instantiated data structures, and method calls as the application of these subprograms to the appropriate data structure.

The main difference between the approach adopted in the implementation of DRAGOON and the simple "abstract access type" approach is that the state of objects is not represented by a single record but by a (linked) *list* of records. Each node in the list stores the state variables added by ancestors of the object's class. Thus the state of a *PERISHABLE_DESCRIPTOR* object would be represented by a list with the "node" storing the *SIMPLE_CALENDAR.DATE* attribute specific to *PERISHABLE_DESCRIPTOR* objects added to a "node" storing the four attributes belonging to *PRODUCT_DESCRIPTOR* objects.

Although it is not reflected explicitly in the specification of classes, as mentioned in chapter 3, conceptually the root of the DRAGOON inheritance hierarchy is the class *APPLICATION_OBJECT*. In other words, all user defined application classes in DRAGOON are descendents of *APPLICATION_OBJECT*, and may be assigned to instance variables of this class. For logical, as well as practical reasons, therefore, the

first node of every such *state list* corresponds to an instance of the class *APPLICATION_OBJECT*. The form of other nodes appended to this list then depends on the position of the object's class in the inheritance hierarchy.

The Ada record type used to generate this special first node in state lists is defined in the following package forming part of the predefined environment of every Ada library used for implementing DRAGOON :-

```
package APPLICATION is                                          /7.6/
    type STATE;
    type OBJECT is access STATE;
    type STATE is record
        OFFSPRING_NO : NATURAL := 0;
        SELF : OBJECT;
        MULTIPLE : OBJECT;
        HEIR : OBJECT;
    end record;
    function CREATE (OFFSPRING_NO : in NATURAL := 0) return OBJECT;
end;

package body APPLICATION is
    function CREATE (OFFSPRING_NO : in NATURAL := 0) return OBJECT is
        OB : OBJECT;
    begin
        OB := new STATE;
        OB.OFFSPRING_NO := OFFSPRING_NO;
        OB.SELF := OB;
        OB.MULTIPLE := null;
        OB.HEIR := null;
        return OB;
    end;
end APPLICATION;
```

All objects in the system are represented by state lists whose first node is a record of type *APPLICATION.STATE*, referenced by an access variable of type *APPLICATION.OBJECT*. Instances of class *APPLICATION_OBJECT* are a special case in that their state is represented by a "uni-node" list containing a single record of this type. The state lists corresponding to descendents of *APPLICATION_OBJECT* are composed of an *APPLICATION.STATE* record, followed by records holding the state variables introduced by each of the classes in the inheritance chain.

This is the basis of the strategy for overcoming the incompatibility of DRAGOON's polymorphism features and Ada's strong typing mechanism. Since the first node of every state list is of the type *APPLICATION.STATE* all objects in the

system are referenced by access values of the same Ada type *APPLICATION.OBJECT*. All of the DRAGOON typing rules related to classes and objects, therefore, have to be checked and enforced by the DRAGOON translation tool, not by the typing rules of Ada.

The fields *OFFSPRING_NO* and *MULTIPLE* of *APPLICATION.STATE* are used to handle multiple offspring and multiple inheritance situations respectively and will be described fully below. The field *SELF*, on the other hand, simply points back to the record so that object may access their own state. The last field *HEIR* is the one which contains the "links" or references to other nodes to build a linked list for non-trivial objects. In the case of instances of the class *APPLICATION_OBJECT* this field is left containing the value "**null**" since the "state" of such objects is represented by an instance of *APPLICATION.STATE* alone. However, for objects of descendant classes this field is used to point to the next node in the list.

In the above package the *HEIR* field of the record *STATE* is defined to be of type *OBJECT* for convenience, but in fact any access type would have sufficed, because it is impossible to predict at the time of its definition what the type of the next node will be. This information is only available when an heir of *APPLICATION_OBJECT* is transformed. This is the point at which the Ada typing rules need to be broken so that the nodes representing newly defined heir classes can be "linked" onto the list corresponding to objects of the parent class. To make the *HEIR* field point to a record with a different type to *APPLICATION.OBJECT* the generic function *UNCHECKED_CONVERSION* conversion must be used to change its apparent type. The use of this facility, however, is strictly confined to two special functions.

Consider the class *PRODUCT_DESCRIPTOR* (3.4), for example. This is transformed into a package with the following specification :-

```
with APPLICATION, PRODUCT;                                        /7.7/
package PRODUCT_DESCRIPTOR is

    type VARIABLES;
    type STATE is access VARIABLES;

    type VARIABLES is record
        INTERNAL_BAR_CODE : PRODUCT.CODE_NUMBER;
        INTERNAL_PRICE : PRODUCT.VALUE;
        INTERNAL_STOCK_CODE : PRODUCT.CODE_NUMBER;
        INTERNAL_LABEL : PRODUCT.LABEL;
        OFFSPRING_NO : NATURAL := 0;
        HEIR : APPLICATION.OBJECT;
    end record;

    function PART_OF (OB : in APPLICATION.OBJECT) return STATE;
    function CREATE (OFFSPRING_NO : in NATURAL := 0)
                                          return APPLICATION.OBJECT;
```

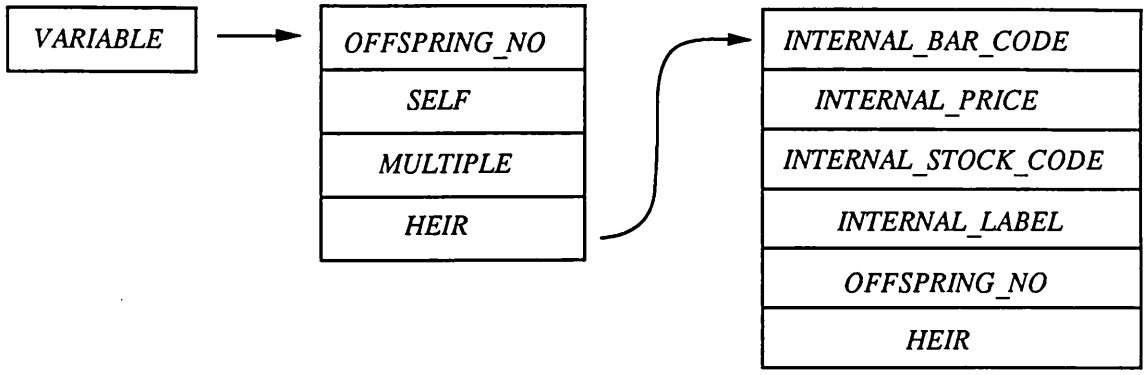


Figure 7.1: List Structure holding State of PRODUCT_DESCRIPTOR objects.

```
        function PRICE (OB : in APPLICATION.OBJECT) return PRODUCT.VALUE;
        procedure SET_PRICE_TO (OB : in APPLICATION.OBJECT;
                                                    PRICE : in PRODUCT.VALUE);
        function BAR_CODE(OB : in APPLICATION.OBJECT)
                                                    return PRODUCT.CODE_NUMBER;
            :
                -- similarly for other "methods" of PRODUCT_DESCRIPTOR
            :
    end PRODUCT_DESCRIPTOR;
```

7.2.1 The CREATE Function

As illustrated in figure 7.1 the state of a *PRODUCT_DESCRIPTOR* object is represented by a linked list with two nodes, the first of type *APPLICATION.STATE* and the second of type *PRODUCT_DESCRIPTOR.VARIABLES* storing the four attributes specific to product descriptor objects. The job of linking the two nodes together transparently is performed by the *CREATE* function exported in the specification.

To enable the *HEIR* field of the first node to point to a record of type *PRODUCT_DESCRIPTOR.VARIABLES* the *CREATE* function has to make references of type *PRODUCT_DESCRIPTOR.STATE* appear to be of the expected type, *APPLICATION.OBJECT*, using *UNCHECKED_CONVERSION*. The body of the *CREATE* function declared in *PRODUCT_DESCRIPTOR* is thus of the form :-

```
        :
    function CREATE (OFFSPRING_NO : in NATURAL := 0)
                                                    return APPLICATION.OBJECT is
```

```
        APP : APPLICATION.OBJECT;
        PROD : PRODUCT_DESCRIPTOR.STATE;
    begin
        PROD := new PRODUCT_DESCRIPTOR.VARIABLES;
        PROD.OFFSPRING_NO := OFFSPRING_NO;
        PROD.HEIR := null;
        APP := APPLICATION.CREATE(1);
        APP.HEIR := APPLICATION_VIEW_OF(PROD);
        return APP;
    end;
        :
```

where *APPLICATION_VIEW_OF* is an instantiation of the generic function *UNCHECKED_CONVERSION* converting access values of type *PRODUCT_DESCRIPTOR.STATE* to the type *APPLICATION.OBJECT* :-

```
        :
    function APPLICATION_VIEW_OF is new UNCHECKED_CONVERSION
                        (PRODUCT_DESCRIPTOR.STATE, APPLICATION.OBJECT);
        :
```

As far as the Ada typing system is concerned, therefore, *PRODUCT_DESCRIPTOR* objects, and indeed all objects in the system, are of type *APPLICATION.STATE* referenced by an access variable of type *APPLICATION.OBJECT*. Not only does this solve the problem of polymorphism, but it means that there are no type obstacles to the incremental introduction of new subclasses of *PRODUCT_DESCRIPTOR*, since instances of these are also represented by state lists referenced by accessed variables of type *APPLICATION.OBJECT*. Nevertheless, it is quite easy to tell at any stage which type of object such a reference actually points to by testing the *HEIR* fields of the nodes in the list. For example, if the *HEIR* field of the first node is "**null**" the list represents an instance of class *APPLICATION_OBJECT*. If not then it must correspond to a descendant of *APPLICATION_OBJECT*, and therefore can be supplied as a parameter to a method of a descendant class.

7.2.2 The PART_OF Function

The *PART_OF* function defined in the package specification performs the inverse "*UNCHECKED_CONVERSION*" to the *CREATE* function so that the "methods" of the class can manipulate the state variables stored in the corresponding state node. Given a reference of type *APPLICATION.OBJECT*, it returns a reference of the access type defined in the package specification - in this case *PRODUCT_DESCRIPTOR.STATE*. The *PART_OF* function declared in *PRODUCT_DESCRIPTOR*, for example, would have a very simple body of the form :-

```
    :
function PART_OF (OB : in APPLICATION.OBJECT) return STATE is          /7.9/
begin
    return PRODUCT_DESCRIPTOR_VIEW_OF (OB.HEIR);
end;
    :
```

where *PRODUCT_DESCRIPTOR_VIEW_OF* is an instantiation of *UNCHECKED_CONVERSION* providing the reverse conversion to *APPLICATION_VIEW_OF* used by the *CREATE* method :-

```
    :
function PRODUCT_DESCRIPTOR_VIEW_OF is new UNCHECKED_CONVERSION
                    (APPLICATION.OBJECT, PRODUCT_DESCRIPTOR.STATE);
    :
```

Apart from these two functions, *CREATE* and *PART_OF*, no other part of the Ada code makes use of *UNCHECKED_CONVERSION*. The breaking of the typing rules is therefore performed "transparently" in a disciplined manner within the bodies of these methods.

7.2.3 Linear Inheritance

In a linear inheritance hierarchy (i.e. classes having only one parent) this process of adding nodes onto the list is repeated for each new addition to the hierarchy. Thus, as illustrated in figure 7.2 the state of a *PERISHABLE_DESCRIPTOR* (3.7) object is stored as a linked list of three nodes - the two nodes corresponding to the *PRODUCT_DESCRIPTOR* part and an extra node for storing the additional *SIMPLE_CALENDAR.DATE* attribute of *PERISHABLE_DESCRIPTOR* objects. The Ada package into which *PERISHABLE_DESCRIPTOR* is translated, however, is completely independent of the record types used to generate the first two nodes in the list. In other words, the transformation of a subclass into Ada does not depend in any way on the nature of the inherited classes, but merely on the form of the class itself. *PERISHABLE_DESCRIPTOR*, for example, would be transformed into package of the form :-

```
with APPLICATION, SIMPLE_CALENDAR;                                     /7.10/
package PERISHABLE_DESCRIPTOR is

    type VARIABLES;
    type STATE is access VARIABLES;

    type VARIABLES is record
        INTERNAL_DATE : SIMPLE_CALENDAR.DATE;
        OFFSPRING_NO : NATURAL := 0;
        HEIR : APPLICATION.OBJECT;
```

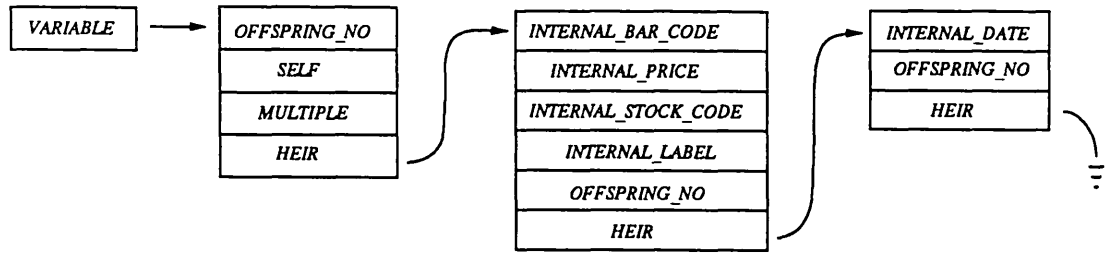


Figure 7.2: List Structure holding State of PERISHABLE_DESCRIPTOR objects.

```
        end record;
        function PART_OF (OB : in APPLICATION.OBJECT) return STATE;
        function CREATE (OFFSPRING_NO : in NATURAL := 0)
                                                  return APPLICATION.OBJECT;
        function SELL_BY_DATE (OB : in APPLICATION.OBJECT)
                                                  return SIMPLE_CALENDAR.DATE;
        function SET_SELL_BY_DATE (OB : in APPLICATION.OBJECT;
                                   DATE : in SIMPLE_CALENDAR.DATE);
    end PERISHABLE_DESCRIPTOR;
```

whose structure is not effected in any way by that of the package *PRODUCT_DESCRIPTOR* corresponding to its parent class. The only place in which reference is made to this package is in the implementation of the *CREATE* and *PART_OF* functions. The *CREATE* function of *PERISHABLE_DESCRIPTOR* uses the *CREATE* function of *PRODUCT_DESCRIPTOR* to generate the appropriate state list :-

```
    function CREATE (OFFSPRING_NO : in NATURAL := 0)                  /7.11/
                                                  return APPLICATION.OBJECT is
        APP : APPLICATION.OBJECT;
        PER : PERISHABLE_DESCRIPTOR.STATE;
    begin
        PER := new PERISHABLE_DESCRIPTOR.VARIABLES;
        PER.OFFSPRING_NO := OFFSPRING_NO;
        PER.HEIR := null;
        APP := PRODUCT_DESCRIPTOR.CREATE(1);
```

```
        PRODUCT_DESCRIPTOR.PART_OF(APP).HEIR
                                        := APPLICATION_VIEW_OF(PER);
        return APP;
    end;
```

where *APPLICATION_VIEW_OF* performs precisely the same kind of conversion as in the above version of *CREATE* (7.2.1), but from *PERISHABLE_DESCRIPTOR.STATE* to *APPLICATION.OBJECT*.

Notice how this *CREATE* function, belonging to *PERISHABLE_DESCRIPTOR*, uses the *CREATE* and *PART_OF* functions corresponding to the parent class, *PRODUCT-_DESCRIPTOR*, to generate a *PRODUCT_DESCRIPTOR* list and then to append the record containing the variables specific to *PERISHABLE_DESCRIPTOR* objects onto the end. The function is not in anyway concerned with the structure of the list returned by the parent's *CREATE* function, but is merely interested in "tagging" a *PERISHABLE_DESCRIPTOR* node onto the end.

Similarly, the *PART_OF* function of *PERISHABLE_DESCRIPTOR* uses the *PART_OF* function exported by its parent's package :-

```
        :
    function PART_OF (OB : in APPLICATION.OBJECT) return STATE is      /7.12/
    begin
        return PERISHABLE_DESCRIPTOR_VIEW_OF
                                    (PRODUCT_DESCRIPTOR.PART_OF(OB).HEIR);
    end;
        :
```

This technique of using the *CREATE* and *PART_OF* functions to append and access new "nodes" to state lists using the corresponding functions of the parent classes lies at the heart of the DRAGOON technique for supporting inheritance and incremental development. The key benefit of this mechanism is that the structure of these two functions, and consequently of the whole package, is not at all influenced by the implementation of the parent class. All classes can therefore be translated according to the same rules irrespective of their position in the inheritance hierarchy.

The Ada code corresponding to methods and threads, which are the only things in DRAGOON which can effect the state of an object, knows the type of the Ada record in which it is stored, and can call the appropriate *PART_OF* function to do the *UNCHECKED_CONVERSION* necessary to access it. All such conversions are guaranteed to be safe since they are only performed if the *HEIR* field of the preceding node is not "**null**". At the same time, however, since all objects are referenced by the same Ada type (*APPLICATION.OBJECT*) no retransformation or recompilation is needed when new classes are added to the system.

7.2.4 Clientship

This method of implementing DRAGOON classes in Ada makes the translation of client code very straightforward. *All* instance variables, of whatever class type, are translated into Ada access variables of type *APPLICATION.OBJECT*, since the first node of all state lists is of type *APPLICATION.STATE*. The translation of method invocations employs the same principle used in the simple "abstract data type" mapping outlined in section 7.1.3 above. In other words, the Ada access variable corresponding to the called object is supplied as the first parameter of the subprogram implementing the method. Thus a method call of the following kind in a client of *PRODUCT_DESCRIPTOR* :-

```
    :
PROD.SET_PRICE_TO(PRICE);
    :
```

where *PROD* is a *PRODUCT_DESCRIPTOR* instance variable, and *PRICE* an *INTEGER* value, would be transformed into the following subprogram invocation :-

```
    :
PRODUCT_DESCRIPTOR.SET_PRICE_TO(PROD, PRICE);
    :
```

where *PROD* is now an access variable of type *APPLICATION.OBJECT*.

Similarly, the generation of objects by invocation of the *CREATE* method is simply translated into a call to the *CREATE* function in the corresponding Ada package :-

```
    :
PROD := PRODUCT_DESCRIPTOR.CREATE;
    :
```

To illustrate these principles in a concrete example consider the class *SIMPLE_QUEUE* (4.1) introduced in chapter 4. This would be translated into an Ada package with the following specification :-

```
package SIMPLE_QUEUE is                                          /7.13/
    type QUEUE_POINTER;
    type QUEUE_NODE is record
        ITEM : APPLICATION.OBJECT;
        NEXT : QUEUE_POINTER;
    end record;
    type QUEUE_POINTER is access QUEUE_NODE;

    type VARIABLES;
    type STATE is access VARIABLES;

    type VARIABLES is record
        FRONT, LAST : QUEUE_POINTER;
```

```
        OFFSPRING_NO : NATURAL := 0;
        HEIR : APPLICATION.OBJECT;
    end record;
    function PART_OF (OB : in APPLICATION.OBJECT) return STATE;
    function CREATE (OFFSPRING_NO : in NATURAL := 0)
                                                  return APPLICATION.OBJECT;
    function POP (OB : in APPLICATION.OBJECT)
                                                  return APPLICATION.OBJECT;
    procedure ADD (OB, ITEM : in APPLICATION.OBJECT);
    function IS_EMPTY (OB : in APPLICATION.OBJECT) return BOOLEAN;
    UNDER_FLOW : exception;
end SIMPLE_QUEUE;
```

The first three types in this package are those used to define the list structure implementing the queue, and appear in the body of the class. In general, any type definitions appearing in the a class specification or body declarative part are simply copied into the specification of the corresponding Ada package.

Note also how all the method parameters and variables which were originally of a class type (i.e. *EXAMPLE* in the case of *SIMPLE_QUEUE*) are translated into the Ada type *APPLICATION.OBJECT*.

7.3 Dynamic Binding

The preceding sections have illustrated how the inheritance and polymorphism features of DRAGOON can be faithfully translated into Ada. The remaining object-oriented feature of DRAGOON most alien to Ada is dynamic binding.

To illustrate the problems involved in implementing this mechanism consider the class *BOUNDED_SIMPLE_QUEUE* (4.3) defined in chapter 4. The most important feature of this class as far as dynamic binding is concerned is that it reimplements some of the methods inherited from *SIMPLE_QUEUE*. Consequently, when one of the redefined methods is invoked through an instance variable of class *SIMPLE_QUEUE* the particular version of the method which is executed depends on the dynamic type of the instance variable - that is, the type of the object to which it is referring at the time of the call. The problem, therefore, is to decide at run-time which of the Ada subprograms implementing the alternative versions of the method should be executed. Moreover, since the incremental development facilities of DRAGOON means that the programmer may define further subclasses at any later stage, the range of different versions which may be invoked does not remain fixed.

7.3.1 Method Selection Shells

Clearly, there must be some Ada code in the system which knows about all the different current versions of a method in the system and is able to select the appropriate version at run-time. If this were embedded in the body of the Ada packages into which classes are translated, however, the code would have to be reproduced and recompiled each time a new version of a method were defined in a subclass. The incremental development principle of DRAGOON, and all the benefits gained by the linked list technique for representing state, would thus be largely undermined.

Ada's facility for defining the bodies of methods in physically separate subunits, however, provides an elegant mechanism for avoiding this problem. It permits the amount of code that has to be updated to cater for the introduction of new method versions to be limited to a single procedure body.

The discussion above has described how the methods of a class are transformed into subprograms exported by the corresponding Ada package. None of the subprograms declared in the package *specification* actually implements the corresponding method *directly*, however. This job is, in fact, performed by an additional set of methods declared in a package, *SELF*, contained in the body of the main package. The bodies of the visible subprograms declared in the specification of the main package are contained in "**separate**" units, and use the subprograms defined in the inner package, *SELF*, to implement the original DRAGOON method.

Consider the class *SIMPLE_QUEUE*, for example. For each of the methods exported by this class there are *two* subprograms in the corresponding Ada code - one declared in the specification of the package *SIMPLE_QUEUE* and the other declared in the body. The body of the Ada package *SIMPLE_QUEUE* therefore has the general form :-

```
package body SIMPLE_QUEUE is                                        /7.14/
    function POP (OB : in APPLICATION.OBJECT)
                                    return APPLICATION.OBJECT is separate ;
    procedure ADD (OB, ITEM : in APPLICATION.OBJECT) is separate ;
    function IS_EMPTY (OB : in APPLICATION.OBJECT)
                                              return BOOLEAN is separate ;

    package SELF is
        function POP (OB : in APPLICATION.OBJECT)
                                              return APPLICATION.OBJECT;
        procedure ADD (OB, ITEM : in APPLICATION.OBJECT);
        function IS_EMPTY (OB : in APPLICATION.OBJECT) return BOOLEAN;
    end SELF;

    package body SELF is
        :
```

```
            -- bodies of the "implementing" methods
        :
    end SELF;
     :
        -- bodies of the CREATE and PART_OF methods
     :
end;
```

Since the subprograms declared in the inner package, *SELF*, contain the Ada image of the code in the body of the corresponding methods, these are termed the "*implementing*" subprograms. The implementing subprogram for the *POP* method, for example, would be of the form :-

```
    :                                                          /7.15/
function POP (OB : in APPLICATION.OBJECT) return APPLICATION.OBJECT is
    FRONT_NODE : QUEUE_POINTER := SIMPLE_QUEUE.PART_OF(S).FRONT;
begin
    if SIMPLE_QUEUE.PART_OF(S).FRONT = null then
        raise UNDERFLOW;
    else
        SIMPLE_QUEUE.PART_OF(S).FRONT :=
                                    SIMPLE_QUEUE.PART_OF(S).FRONT.NEXT;
        return FRONT_NODE;
    end if;
end;
    :
```

In addition to the translation of clientship dependencies in the manner described above, the main change occurring in the translation of a DRAGOON method body into the corresponding "implementing subprogram" is the extension of instance variable and actual parameter names by the *PART_OF* function to provide access the appropriate node of the state list. A similar translation is performed for all the other methods introduced by *SIMPLE_QUEUE*.

Since *SIMPLE_QUEUE* is an heir only of *APPLICATION_OBJECT*, and does not inherit any user-defined method, when it is first translated into Ada there is only one version of each of its methods known to the system. Until a subclass of *SIMPLE_QUEUE* is added to the library, therefore, the subprograms declared in the specification of the corresponding Ada package are essentially redundant. The only action they perform is to call the corresponding method contained in *SELF*. The body of the exported *POP* subprogram, for example, has a *separate* body of the form :-

```
separate (SIMPLE_QUEUE)                                       /7.16/
function POP (OB : in APPLICATION.OBJECT) return APPLICATION.OBJECT is
begin
```

```
        return SELF.POP(SIMPLE_QUEUE.PART_OF(OB));
    end;
```

At this stage, therefore, this subprogram makes no useful contribution to the implementation of the method. It is when the programmer defines new versions of the method in subclasses of *SIMPLE_QUEUE* that the purpose of these subprograms becomes apparent. One subclass of *SIMPLE_QUEUE* which does just this is the class *BOUNDED_SIMPLE_QUEUE* which redefines the *CREATE*, *POP* and *ADD* methods of *SIMPLE_QUEUE* to provide a bounded version of the component according to the implementation defined in (4.3).

When subclasses such as this are translated into Ada, methods *introduced* or *redefined* by the class (i.e. those for which it must provide a body) are translated according to the principles just described. In other words, each such method is translated into two Ada subprograms, one exported in the specification of the corresponding package, and the other "implementing method" defined in the inner package *SELF*. No methods are produced, however, for methods such as *IS_EMPTY* which are inherited by the subclass, but not redefined.

However, the generation of the corresponding Ada package is not the only action performed during the translation of a class which redefines inherited methods such as *POP*. In addition, the translation tool updates, or replaces, the subunits (i.e. the bodies) of the subprograms exported by the parent class's package, in order to take account of the new version of the method introduced into the system. When *BOUNDED_SIMPLE_QUEUE* is translated into Ada, for example, the separate body of the subprogram *SIMPLE_QUEUE.POP* illustrated above is replaced by the following :-

```
separate (SIMPLE_QUEUE)                                          /7.17/
function POP (OB : in APPLICATION.OBJECT) return APPLICATION.OBJECT is
begin
    if SIMPLE_QUEUE.PART_OF(OB) = null then
        return SELF.POP(SIMPLE_QUEUE.PART_OF(OB));
    else
        return BOUNDED_SIMPLE_QUEUE.POP(OB);
    end if;
end;
```

When invoked, this function analyses the form of the state list, *OB*, to see whether the node corresponding to *BOUNDED_DESCRIPTOR_QUEUE* is present (i.e. whether the list represents an instance of *DESCRIPTOR_QUEUE* or its subclass *BOUNDED_DESCRIPTOR_QUEUE*), and if not (i.e. it represents the former) it invokes the *DESCRIPTOR_QUEUE* implementation, otherwise it invokes the *BOUNDED_DESCRIPTOR_QUEUE* version. Essentially, therefore, it forms a kind of "shell" around the true

method implementations in order to select, at run-time, the appropriate one for execution according to the dynamic binding rules of DRAGOON. Such a subprogram is thus termed a "*selection shell*" for the method concerned.

The great advantage of this approach is that because the method body replaced is a subunit of the main package, not even the body of the package, let alone the Ada code for clients of the class need to be recompiled when method-redefining subclasses are added to the system. The client Ada code may therefore exploit the new version of the method without any modification. All the modification and recompilation needed to cater for the new version is limited to the selection shells of the methods concerned.

7.3.2 Multiple Offspring

This technique is fine for distinguishing between the different versions of a method that may be introduced in a linear inheritance chain - that is, when each class has only one parent and one heir. In general, however, neither of these two conditions need be satisfied since the multiple inheritance facilities of DRAGOON (whose implementation is discussed in the next section) permit classes to have more than one parent, while it is a normal feature of most object-oriented languages that a class may have more than one heir, or *offspring.*

Clearly, merely checking the *HEIR* field of a state node is not sufficient to indicate which of the numerous offspring of the corresponding class the state list represents. If the *HEIR* field of a *PRODUCT_DESCRIPTOR* (3.4) node was not "**null**", for example, the next node could either correspond to a *PERISHABLE_DESCRIPTOR* (3.6) object or to an *ELECTRICAL_DESCRIPTOR* (3.8) object. Before using a *PART_OF* function to convert its type it is essential to determine which of the subclasses the next node in the list actually corresponds to, otherwise serious conversion errors could result. Another field is needed in the nodes of the state lists, therefore, to indicate which of the potential offspring the subsequent node represents - that is, which of the branches of the inheritance tree the class represented by the subsequent node lies in.

There are a number of possible ways of storing this information. One way is simply to store the class name. For example, the record type defining the node corresponding to *PRODUCT_DESCRIPTOR* could contain a string field holding either "PERISHABLE_DESCRIPTOR" or "ELECTRICAL_DESCRIPTOR" depending on the nature of the following node. The current version of the translation tool, however, has adopted an alternative strategy in which an integer identifier, called the "offspring number", is assigned to the heirs of a class according to the order in which they are submitted to the translation tool. This is the purpose of the *OFFSPRING_NO* field

introduced above.

If *PERISHABLE_DESCRIPTOR* were the first successfully compiled subclass of *PRODUCT_DESCRIPTOR*, for example, it would be assigned the offspring number, 1, with respect to *PRODUCT_DESCRIPTOR*. In other words, in a state list corresponding to a *PERISHABLE_DESCRIPTOR* object the *OFFSPRING_NO* field of the *PRODUCT_DESCRIPTOR* (i.e. the second) node would contain the value, 1. The next successfully translated classes would be assigned the offspring number, 2, with respect to *PRODUCT_DESCRIPTOR* and so on.

Although this limits the number of offspring of a particular class to the value of *MAX_INT* this is not felt to be a practical constraint on the development of class libraries. The advantage of this approach is that it reduces the size of the state list nodes, and also the time taken to test the equality of different offspring identifiers.

Together, the *HEIR* and *OFFSPRING_NO* fields of state nodes provide all the information needed by selection shells to determine which version of a method to execute in response to a call. Suppose, for example, that the programmer defined another subclass of *SIMPLE_QUEUE*, *ANOTHER_SIMPLE_QUEUE* say, which also redefined the *POP* method. Assuming that this was translated *after* the *BOUNDED_SIMPLE_QUEUE*, and that there are no other subclasses, *ANOTHER_SIMPLE_QUEUE* would be assigned the offspring number, 2, and *BOUNDED_SIMPLE_QUEUE*, 1. In order to choose the appropriate implementation when the *POP* method is invoked through an instance variable of class *SIMPLE_QUEUE*, the body of the *POP* selection shell would be updated to :-

```
separate (SIMPLE_QUEUE)                                          /7.18/
function POP (OB : in APPLICATION.OBJECT) return APPLICATION.OBJECT is
begin
    if SIMPLE_QUEUE.PART_OF(OB) = null then
        return SELF.POP(OB);
    else
        case OFFSPRING_NO is
        when 1 =>
            return BOUNDED_SIMPLE_QUEUE.POP(OB);
        when 2 =>
            return ANOTHER_SIMPLE_QUEUE.POP(OB);
        when others =>
            return SELF.POP(OB);
        end case ;
    end if;
end;
```

Notice that if the *HEIR* field checked by a selection shell is not "**null**", and a version of the method defined by a subclass is required, it does not directly call

the required implementing subprogram (in fact, it can not because it is hidden in a package body) but rather the *selection shell* corresponding to the subclass indicated by the *OFFSPRING_NO* field. A selection shell, therefore, does not in general choose between *all* the available versions of a method, but merely decides whether to call the version defined by the associated class (implemented by the subprogram contained in the inner *SELF* package) or whether to call a *selection shell* of one of the class's descendents. In the second case, the called selection shell continues the selection process in the same way. The call hierarchy of the selection shells, therefore, directly corresponds to the inheritance hierarchy of the associated classes.

This means that the time taken to select between the various versions of a method is not constant, but is proportional to the actual number of different versions. This, in turn, is proportional to the depth of the inheritance hierarchy. Since there is in principle no limit to the depth of class inheritance hierarchies, and hence the number of potential versions of a method that could be executed in response to a method call, the time taken to select the version required is potentially unbounded. Experience has demonstrated, however, that inheritance hierarchies developed in practical programming situations tend to remain fairly shallow, and the number of times a method is redefined is fairly small. The time taken to search for the appropriate method body to execute, therefore, is unlikely to present significant difficulties in most applications. Nevertheless, it is important that applications programmers are aware of the mechanism used to search for method versions so that they may avoid the overheads involved in time critical situations.

One solution might be to include an additional keyword in the language which programmers could use in such situations to indicate that the invocation time for a particular method is more important than the dynamic binding facility. In other words, the keyword - **"fixed"** or **"immutable"**, say - would indicate that the corresponding method can not be redefined in subclasses, and that its access time is therefore fixed.

7.3.3 Abstract Classes

Since the nodes in the state lists are not used just for storing the state variables of the corresponding classes, but also for representing the position of the corresponding object in the inheritance hierarchy, a state list for an object of a given class must contain nodes for every ancestor of the class, even if they introduce no additional instance variables. In particular, this means that nodes are needed for abstract classes no matter whether they are completely "abstract" and define no state variables or method implementations.

Abstract classes are, in fact, translated into packages with precisely the same

structure as those produced for normal classes. The only difference is that the implementing subprograms corresponding to deferred methods simply contain a **raise** statement to raise the appropriate exception should it be erroneously invoked. The packages corresponding to abstract classes even contain a *CREATE* subprogram, since this is needed by the *CREATE* subprograms of descendent classes in order to generate the appropriate state list. The "concreting" of deferred methods in subclasses is handled in precisely the same way as the redefinition of normal methods.

7.4 Multiple Inheritance

The discussion so far has been concerned with translating classes possessing a maximum of one parent. In common with a number of other object-oriented languages, however, DRAGOON permits classes to inherit the properties of multiple parents. The extra possibilities which this introduces, particularly the range of different assignments that it makes possible, significantly increases the complexity of the translation into Ada because it means that the straightforward linked list structure is no longer adequate for storing the state of objects.

To see why this is so, consider the class *DATED_DESCRIPTOR* (3.13) introduced in chapter 3 as an heir of both *PRODUCT_DESCRIPTOR* (3.4) and *DATED_ITEM* (3.12). Since it is a normal (although perhaps somewhat trivial class) *DATED_ITEM* will be transformed, according to the principles outlined above, into an Ada package with the following specification part :-

```
with APPLICATION, SIMPLE_CALENDAR;                                    /7.19/
package DATED_DESCRIPTOR is

    type VARIABLES;
    type STATE is access VARIABLES;

    type VARIABLES is record
        INTERNAL_DATE : SIMPLE_CALENDAR.DATE;
        OFFSPRING_NO : NATURAL := 0;
        HEIR : APPLICATION.OBJECT;
    end record;

    function PART_OF (OB : in APPLICATION.OBJECT) return STATE;
    function CREATE (OFFSPRING_NO : in NATURAL := 0)
                                              return APPLICATION.OBJECT;

    function DATE (OB : in APPLICATION.OBJECT)
                                            return SIMPLE_CALENDAR.DATE;
    procedure SET_DATE_TO (OB : in APPLICATION.OBJECT;
                                        DATE : in SIMPLE_CALENDAR.DATE);
end;
```

The main problem presented by multiple inheritance is the fact that a class with multiple parents does not have a unique ancestry chain, and cannot therefore be represented by a single list. Consider an instance of the class *DATED_DESCRIPTOR*, for example. Since this a subclass of *PRODUCT_DESCRIPTOR* it can be assigned to instances variables of this class type, and can respond to the same set of method invocations as instances of *PRODUCT_DESCRIPTOR*. In terms of the Ada implementation described above this means that the state list representing this instance can be supplied to the subprograms defined in the *PRODUCT_DESCRIPTOR* package, which expect the first part of the list to be the same as that for *PRODUCT_DESCRIPTOR* objects.

By the same token, however, since *DATED_DESCRIPTOR* is a subclass of *DATED_ITEM* also, instances can be supplied to *DATED_ITEM* instances variables. In terms of the Ada transformation, this means that the state list for such an object can be assigned to subprograms which expect the first part of the list to be that of a *DATED_ITEM* object. Clearly, it is impossible for a single list to be both a *PRODUCT_DESCRIPTOR* list and *DATED_ITEM* list at the same time. In order to provide this apparent duality in the translation of DRAGOON, instances of classes with multiple parents are not represented by a single state list, but by a composite data structure containing all the necessary state lists. Provided that from any instance variable, the appropriate list, or *form*, can be reached, the Ada subprograms implementing the methods of parent classes need only be supplied with the list they are interested in.

Perhaps not surprisingly, it is the role of the one remaining field of the *APPLICATION.STATE* record to link together the necessary lists into the required composite data structure. When more than one state list is needed to store the state of an object because of multiple inheritance, the *MULTIPLE* field in the first node in each list is made to point to the first node of the next list (except of course the last, which point to the first list). The result, therefore, is perhaps best described as a "ring" of linked lists.

The data structure for storing the state of *DATED_DESCRIPTOR* objects is represented in figure 7.3. As indicated in this figure one of the lists in the ring structure is given a special status in that all the *SELF* fields in the first nodes reference the first node of this list. This list, and the corresponding class, is known as the *default* form of the ring structure, since it is this list which is referenced when an object is created by the *CREATE* function for the class. The default form is arbitrarily chosen to be the first class named in the "inherits" clause of the class being transformed.

Since the class *DATED_DESCRIPTOR* inherits the methods of both its parents, it is possible to invoke the methods inherited both from *PRODUCT_DESCRIPTOR* and

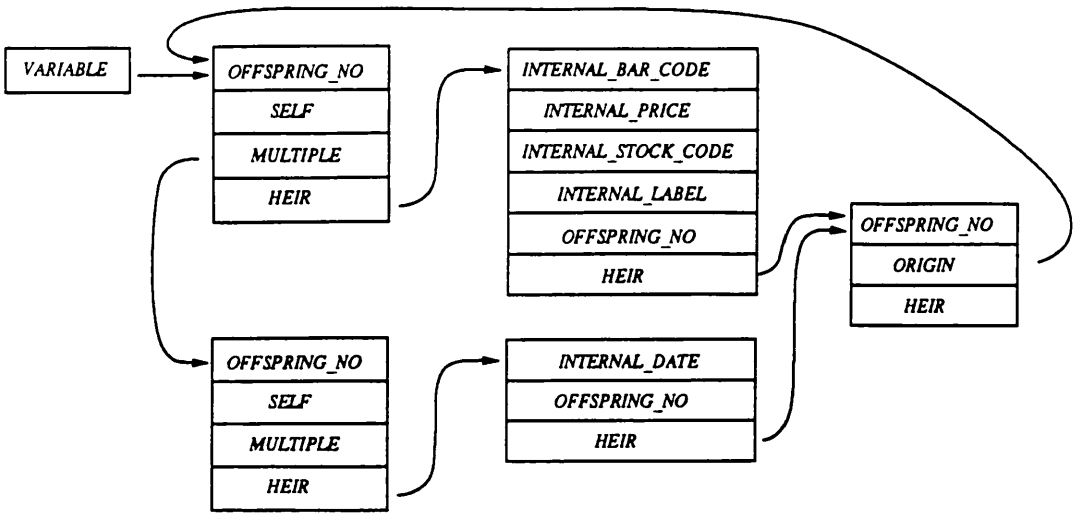


Figure 7.3: Data Structure Storing the State of DATED_DESCRIPTOR Objects.

DATED_ITEM through an instance variable of this class. When translated into Ada, this means that from an access variable corresponding to an instance variable of class *DATED_ITEM* it must be possible to provide (or reach) references to all the different state lists in the structure. As indicated in figure 7.3, from an access variable *DD* referring to the "*PRODUCT_DESCRIPTOR*" list the "*DATED_ITEM*" equivalent can be reached by dereferencing the *MULTIPLE* field of the first node (i.e. by *DD.MULTIPLE*). For classes that inherit from more than two classes the ring structure of lists is simply extended. In other words, the *MULTIPLE* field of the second list would not point back to the first, but would point to the next list in the ring. Therefore, extra dereferencing on the *MULTIPLE* fields is sometimes required to reach other lists in the structure (eg. *DD.MULTIPLE.MULTIPLE.* .....).

By dereferencing the *MULTIPLE* fields the appropriate number of times, therefore, it is possible to reach any of the other forms of an instance of a class with multiple parents from a reference to the default form. In many object-oriented languages this would be sufficient, because if there is only one remaining reference to an object, and this points to one of the forms, the typing rules of the language would imply that the other forms could never be reached anyway. Suppose, for example, that an instance of the class *DATED_DESCRIPTOR* were "pushed" onto an instance of *SIMPLE_QUEUE*. In languages like Eiffel, which make no provision for type conversion, the typing rules mean that this object can only be treated as (i.e. assigned to instance variables) of the class *PRODUCT_DESCRIPTOR*. As described in chapter 3, however, DRAGOON permits the type of the object to be explicitly converted back to *DATED_DESCRIPTOR*, where it can also be treated as a *DATED_ITEM* again.

In general, therefore, it is important not only to be able to reach any list in the structure from the default form, but also to reach the default form from a reference to any of the other lists. The first approach which springs to mind for achieving this is simply to continue dereferencing the *MULTIPLE* fields until the the complete ring structure is traversed, and the original reference is regained. While this clearly works for this simple example, this is not an acceptable solution in general, because when more complex classes are translated, which have multiple parents with multiple parents etc., the size of the ring, and hence the number of dereferences required changes from one class to another.

To permit the reference to the default list to be gained from any of the other list in a ring, as indicated in figure 7.3 the node defined for classes with multiple parents contains an extra field, called *ORIGIN*, which contains a reference back to the default form of the structure. The default form can be reached at any point simply by dereferencing the *ORIGIN* field of the final node in the list. To relieve the client Ada code from having to explicitly express this dereferencing at all points where it is required, additional functions are exported in the interface of the package corresponding to a class with multiple parents, in order to perform this function. More specifically, two additional functions are provided for each parent apart from the first - one to provide a reference to the parents state list from a reference to the default form, and the other to provide the reverse transition.

The DRAGOON class *DATED_DESCRIPTOR* is therefore translated into an Ada package of the following form:

```
with APPLICATION, PRODUCT_DESCRIPTOR, DATED_ITEM;                    /7.20/
package DATED_ITEM is

    type VARIABLES;
    type STATE is access VARIABLES;

    type VARIABLES is record
        OFFSPRING_NO : NATURAL := 0;
        ORIGIN : APPLICATION.OBJECT;
        HEIR : APPLICATION.OBJECT;
    end record;

    function PART_OF (OB : in APPLICATION.OBJECT) return STATE;
    function CREATE (OFFSPRING_NO : in NATURAL := 0)
                                                  return APPLICATION.OBJECT;

    function FROM_DATED_ITEM (OB : in APPLICATION.OBJECT)
                                                  return APPLICATION.OBJECT;
    function DATED_ITEM_FORM (OB : in APPLICATION.OBJECT)
                                                  return APPLICATION.OBJECT;
end;
```

The *FROM_DATED_ITEM* is the function which, given a reference to the *DATED_ITEM*

list in the ring structure returns a reference to the default form using the *ORIGIN* field of the final node. Its body is thus of the form :-

```
    :
function FROM_DATED_ITEM (OB : in APPLICATION.OBJECT
                                    return APPLICATION.OBJECT is
begin
    return DATED_DESCRIPTOR_VIEW_OF
                        (DATED_ITEM.PART_OF(OB).HEIR).ORIGIN;
end;
    :
```

This function is only used in the translation of explicit type conversions of DRAGOON. The other function, *DATED_ITEM_FORM*, which provides a reference to the *DATED_ITEM* list in the structure, given a reference of the default form, is much more commonly employed, and also simpler to implement :-

```
    :
function DATED_ITEM_FORM (OB : in APPLICATION.OBJECT
                                    return APPLICATION.OBJECT is
begin
    return OB.MULTIPLE;
end;
    :
```

7.5 Generic Classes

As described in chapter 4, in common with Eiffel and POOL, DRAGOON supports the notion of generic classes which may be parameterised with respect to data types or class types. Generic classes are surprisingly easy to implement in Ada because of the language's own advanced generic facilities, and also because of the policy that instances of actualisations of generic classes are not type compatible with any other class.

This means that it is not necessary to tackle the problems of polymorphism, dynamic binding and incremental program development encountered in the implementation of normal classes and objects. Consequently, there is no need for the elaborate linked list mechanisms used to permit instances of subclasses to be treated like instances of their parents, or the selection shell mechanisms needed to handle the dynamic selection of method versions. In fact, the state of instances of generic actualisations can be represented using the basic "abstract data type" approach outlined in section 7.1., enhanced to provide the required reference semantics. The main problem encountered in the translation of generic classes is not surprisingly concerned with the generic parameters.

A generic class whose parameters are only of data types can be translated into a generic Ada package with precisely the same generic formal part as the class. Moreover, an actualisation of the generic class is translated into an instantiation of the corresponding Ada package with precisely the same allocation of actual to formal parameters. The type and subprograms exported by the resulting generic actualisation can be used just as any other enhanced "abstract data type" to dynamically generate state records, and perform operations upon them.

In the case of generic classes with one or more parameters of a class type, the generic class is again translated into a single generic package, but the generic class parameters are mapped into a generic type parameter together with a set of associated formal subprograms corresponding to the methods of the formal class type. In other words, the Ada package generated in the translation is parameterised with respect to the same *data abstraction* as the corresponding class, but represented as a type and associated formal subprograms rather than as a formal class parameter.

Consider, for example, the generic class *PAR_CHRONO_QUEUE* (4.16) introduced in chapter 4. This has the class *DATED_ITEM* as a parameter, and is therefore translated into a generic package with the following generic formal part :-

```
generic                                                          /7.21/
    type T is private;
    with function DATE(OB : in T) return SIMPLE_CALENDAR.DATE;
    with procedure SET_DATE_TO (OB : in T,
                                DATE : in SIMPLE_CALENDAR.DATE);
    with function CREATE(OFFSPRING_NO : in NATURAL) return T;
package PAR_CHRONO_QUEUE is

    type STATE is private;
    type OBJECT is access STATE;

    function POP (OB : in APPLICATION.OBJECT) return OBJECT;
        :
            -- similarly for other methods
        :
private
    :
end;
```

Notice the way in which instances generated from actualisations of this class are represented by single records in the basic "abstract access type" style rather than as linked lists. Furthermore, the subprograms exported in the interface are not selections shells, but "implementing" methods whose bodies contain the image of the method bodies.

An actualisation of this class of the form :-

```
class DATED_SIMPLE_QUEUE is new PAR_CHRONO_QUEUE
                                (DATED_ITEM => DATED_DESCRIPTOR);
```

would then be translated into the following instantiation of the generic :-

```
package DATED_SIMPLE_QUEUE is new PAR_CHRONO_QUEUE
                              (T => APPLICATION.OBJECT,
                              DATE => DATED_ITEM.DATE,
                              SET_DATE_TO => DATED_ITEM.SET_DATE_TO,
                              CREATE => DATED_ITEM.CREATE);
```

This examples illustrates how actualisation of *PAR_CHRONO_QUEUE*, with the actual parameter *DATED_DESCRIPTOR* (3.13), would be translated into Ada. Since all subclasses of the formal generic parameter *DATED_ITEM* export at least these methods, the corresponding subprograms can clearly be used to instantiate the generic package.

7.5.1 Inheriting Genericity

One of the novel features of DRAGOON is that it permits generic classes to be specialised by inheritance. In other words, new generic classes can be created by enriching existing generic classes in the same way as normal classes. An important difference, however, resulting from the typing rules associated with generics, is that the actualisations of a generic *subclass* are not type-compatible with actualisation of its generic *superclass*.

Again, therefore, there is no need to worry about the problems of dynamic binding and polymorphism associated with the translation of normal classes. The state of actualisations of generic subclasses can therefore also be modelled by the extended "abstract data type" approach outlined in section 7.1. The new problem introduced by inheriting genericity is to arrange for the inherited state to be incorporated into the state records of instances of generic subclass actualisations, whilst at the same time ensuring that the parameters in the inherited generic formal part remain generic.

This is achieved by actualising the generic package corresponding to the parent class within the package corresponding to the subclass, using the generic formal parameters of the latter as the actual parameters of the former. Consider, for example, the generic class *EXTENDED_CHRONO_QUEUE* (4.22) which is a subclass of *PAR_CHRONO_QUEUE* (4.21) whose translation into Ada is illustrated above. *EXTENDED_CHRONO_QUEUE* would be translated into the following package :-

```
generic                                                              /7.22/
    type INHERITED_T is private;
    with function INHERITED_DATE(OB : in T)
                                              return SIMPLE_CALENDAR.DATE;
    with procedure INHERITED_SET_DATE_TO (OB : in T,
                                          DATE : SIMPLE_CALENDAR.DATE);
    with function INHERITED_CREATE(OFFSPRING_NO : in NATURAL)
                                                          return T;
package EXTENDED_CHRONO_QUEUE is
    package INHERITED is new PAR_CHRONO_QUEUE (T => INHERITED_T,
                                DATE => DATE,
                                SET_DATE_TO => INHERITED_SET_DATE_TO,
                                CREATE => INHERITED_CREATE);
    type STATE is private;
    type OBJECT is access STATE;
    function POP (OB : in APPLICATION.OBJECT) return OBJECT;
        :
            -- similarly for inherited other methods
        :
    function NUMBER_WITH_DATE (OB : in APPLICATION.OBJECT,
                               D : in SIMPLE_CALENDAR.DATE) return NATURAL;
private
    type STATE is record
        STATE : INHERITED.OBJECT;
    end record;
end;
```

In this particular example, the generic subclass *EXTENDED_CHRONO_QUEUE* does not introduce any more generic parameters. It is only generic with respect to those parameters it has inherited. There is no reason why a generic subclass can not introduce new parameters, however, in which case they are transformed in the same way as the parameters of directly defined generic classes.

The same is true of any additional instance variables the generic class may introduce. Had *EXTENDED_CHRONO_QUEUE* introduced any new state variables, these would have simply been included in the private record type *STATE*.

By using the formal generic parameters of the package *EXTENDED_CHRONO_QUEUE* as the actual parameters of the instantiation, *INHERITED*, of *PAR_CHRONO_QUEUE*, the genericity of the inherited state is maintained, but can also be added to the state record corresponding to the generic subclass.

7.6 Concurrency

Having described how the "sequential" features of DRAGOON are implemented in Ada, it is now time to consider the concurrent aspects. These may be conveniently split into two parts - the handling of "active" objects, and the handling of behavioral inheritance, since these are orthogonal features of DRAGOON.

Active Objects

As the task is the unit of concurrency in Ada the implementation of active objects, with their concurrent execution threads, must clearly be based on the use of tasks. The only problem in using a task to represent the thread of an object is in integrating it with the state list representation of objects used so far. In fact, this turns out to be fairly straightforward because Ada permits tasks generated from tasks types to be identified by access variables that can be included in the appropriate record structure. In addition to the fields storing the state variables of the object, therefore, the state node of active objects has an additional field holding a reference to a task.

Consider the active class *FORWARDER* (5.8), for example. This would be translated into a package with a specification of the form :-

```
package FORWARDER is                                                    /7.23/
    type VARIABLES;
    type STATE is access VARIABLES;
    type THREAD;
    type THREAD_REF is access THREAD;
    type VARIABLES is record
        DB, P, BUFFER : APPLICATION.OBJECT;     -- variables of FORWARDER
        OFFSPRING_NO : NATURAL := 0;
        HEIR : APPLICATION.OBJECT;
        THREAD : THREAD_REF;
    end record;
    task type THREAD is
        entry START (S : in APPLICATION.OBJECT);
    end;
    function PART_OF (OB : in APPLICATION.OBJECT) return STATE;
    function CREATE (OFFSPRING_NO : in NATURAL := 0) return
                                                    APPLICATION.OBJECT;
    procedure START (OB : in APPLICATION.OBJECT);
    function SHARE (OB, SHARED_DB : in APPLICATION.OBJECT);
    procedure ADD (OB, P : in APPLICATION.OBJECT);
end;
```

The procedure *START* corresponds to the *START* method used to activate the thread of active classes. Invocation of the *START* method by a client of an active object is thus translated into the invocation of the *START* procedure, whose body is of the form :

```
    :
procedure START (OB : in APPLICATION.OBJECT) is                    /7.24/
begin
    FORWARDER.PART_OF(OB).THREAD := new THREAD;
    FORWARDER.PART_OF(OB).THREAD.START(OB));
end;
    :
```

The first action performed by this procedure is to instantiate the task type, *THREAD*, and assign its access value to the *THREAD* field of the state node associated with *FORWARDER*. The procedure then calls the *START* method of the task to provide the reference to the state list, so that the thread may manipulate the variables of the object. The calling of this entry also serves to unbock the task so that it may begin execution of the code corresponding to the body of the thread. The *THREAD* task type therefore has the following body :-

```
    :
task body THREAD is                                                /7.25/
    SELF : FORWARDER.STATE;
begin
    accept START (S : in APPLICATION.OBJECT) do
    begin
        SELF := FORWARDER.PART_OF(S);
    end;
    SELF.BUFFER := QUEUE.CREATE;
    loop
        SELF.P := GUARDED_QUEUE.POP(SELF.BUFFER);
        DATA_BASE.ADD(SELF.DB, SELF.P);
    end loop;
end;
    :
```

Once the *START* procedure has generated an instance of task type *THREAD*, and provided it with a reference to the object's state list by calling its *START* entry, the task will execute concurrently with other threads, and method invocations, as required. Moreover, as it is included in the state node of the corresponding object, it is intimately associated with the corresponding object state for the duration of the program.

7.6.1 Behavioured Classes

The implementation of behavioured classes in Ada has some similarities to that of active classes in that an additional task is added to the state node corresponding to the class. Unless the behaviour contains a guarded method (section 7.5.2), however, the task implementing the behaviour does not require visibility of the state list of the object. Its function is to export a set of methods which are called before and after the execution of the "implementation" subprograms to enforce adherence to the specified access protocol.

The task types used for this purpose are contained in predefined packages corresponding to the behavioural classes. In the long term tools will be available to generate such packages automatically in the same way as the other parts of the implementing Ada code, but in the short term these package are supplied as part of the library of behavioural classes.

The package corresponding to the behavioural class *READERS_WRITERS* (5.5) for example, which describes the typical "multiple readers - single writers" access protocol, is as follows :-

```
package READERS_WRITERS is                                        /7.26/
    task type BEHAVIOUR is
        entry ROP_REQUEST;
        entry ROP_COMPLETION;
        entry WOP_REQUEST;
        entry WOP_COMPLETION;
    end BEHAVIOUR;
end READERS_WRITERS;

package body READERS_WRITERS is
    task body BEHAVIOUR is
        READERS, WRITERS : NATURAL := 0;
    begin
        loop
            select
                when WRITERS = 0 =>
                    accept ROP_REQUEST do
                        READERS := READERS + 1;
                    end;
            or
                when WRITERS = 0 and READERS = 0 =>
                    accept WOP_REQUEST do
                        WRITERS := WRITERS + 1;
                    end;
            or
```

```
                accept ROP_COMPLETION do
                        READERS := READERS − 1;
                end;
            or
                accept WOP_COMPLETION do
                    WRITERS := WRITERS − 1;
                end;
            end select ;
        end BEHAVIOUR;
    end READERS_WRITERS;
```

For every formal method set appearing in the behavioural class, there are two entries in the corresponding task type - one used to request permission to execute the associated method, and the other to indicate completion of the method.

Suppose that a behavioured version of the *SIMPLE_QUEUE* class discussed above were required, for example. Such a class would be represented by the following behavioured class :-

```
behavioural class SAFE_SIMPLE_QUEUE is                          /7.27/
    inherits SIMPLE_QUEUE;
    ruled by READERS_WRITERS;
where
    ADD => WOP;
    POP => ROP;
end SAFE_SIMPLE_QUEUE;
```

and would be translated into a package with the following specification :-

```
with APPLICATION, READERS_WRITERS                               /7.28/
package SAFE_SIMPLE_QUEUE is

    function PART_OF (OB : in APPLICATION.OBJECT)
                                        return READERS_WRITERS.BEHAVIOUR;
    function CREATE return APPLICATION.OBJECT;

end;
```

Since a behavioured class such as this constitutes the end of the normal inheritance chain, there is no need for the *OFFSPRING_NO* and *HEIR* fields usually included in the state record for classes. Furthermore, since it is not possible to introduce new instance variables in the behavioural inheritance step, there is no need for such fields either. In fact, conceptually the only "field" needed in the state node for a behavioured class is a reference to an instance of a task defining the appropriate behaviour. Under such circumstances, therefore, the record essentially "collapses" into a single access variable referenced by the *HEIR* field of the *SIMPLE_QUEUE* state

list. This access variable is consequently redundant, since the *HEIR* field could easily refer directly to the task rather than to an intermediate access variable.

For this reason no state record is defined in the package generated from a behavioured class. Instead, the *CREATE* method simply generates an instance of the appropriate "behaviour" task (in this case *READERS_WRITERS.BEHAVIOUR*, and appends it to the end of the state list for the application superclass. The *PART_OF* function accordingly returns a reference to this task.

Updating Selection Shells

The crucial part of the translation of behavioured classes is the updating of the selection shells of the affected methods. As well as producing the package illustrated above, when the behavioured class *SAFE_SIMPLE_QUEUE* is translated the selection shells corresponding to the methods inherited from the application class are updated as if they had been redefined in the behavioured class. The difference is that the case statement alternative corresponding to the behavioured class invokes the "request", and "completion", methods of the "behaviour" task before, and after, the implementing method in the package *SELF*.

Consider, for example, the *POP* method of *SIMPLE_QUEUE*. Assuming this class is assigned the "offspring number", 3, with respect to *SIMPLE_QUEUE*, then because *POP* is allocated to the abstract set *ROP* in the behavioured class *SAFE_SIMPLE_QUEUE*, the *POP* selection shell of this class would be updated as follows :-

```
separate (SIMPLE_QUEUE)                                          /7.29/
function POP (OB : in APPLICATION.OBJECT) return APPLICATION.OBJECT is
begin
    if SIMPLE_QUEUE.PART_OF(OB) = null then
        return SELF.POP(SIMPLE_QUEUE.PART_OF(OB));
    else
        case OFFSPRING_NO is
        when 1 =>
            return BOUNDED_SIMPLE_QUEUE.POP(OB);
        when 2 =>
            return ANOTHER_SIMPLE_QUEUE.POP(OB);
        when 3 =>
            SAFE_SIMPLE_QUEUE.PART_OF(OB).ROP_REQUEST;
            return SIMPLE_QUEUE.POP(OB);
            SAFE_SIMPLE_QUEUE.PART_OF(OB).ROP_COMPLETION;
        when others =>
            return SELF.POP(SIMPLE_QUEUE.PART_OF(OB));
        end case ;
    end if ;
end;
```

If the selection shell is applied to the state list for a behavioured object, before the actual implementing subprogram is executed, permission to do so must first be gained from the appropriate "request" entry of the associated "behaviour" task. This is referenced by the *HEIR* field of the last state node. Similarly, once the execution of the implementation method is completed, the appropriate "completion" entry is called to update the state of the "behaviour" task. The client Ada code, therefore, is blocked until the required logical conditions for execution of the corresponding method are satisfied.

7.6.2 Guarded Permissions

This strategy for implementing the synchronisation conditions specified in a behavioural class can easily be extended to cope with situation in which the conditions defining the execution permission of methods are determined not only by the history functions, but also by other methods. The main modification to the translation just described is to allow subprograms in the sequential class's Ada package to affect the acceptance of entry calls in the "behaviour" task, and to give this task visibility of the state list of the object it is protecting.

As described in chapter 5, the methods used as guards on the right hand side of deontic expressions are restricted to being boolean functions. The "behaviour" task type, therefore, defining the entries that must be called before and after executing the implementing methods, can be generalised with respect to the guard methods using generics.

Consider the behavioural class *BOUNDED_BUFFER* (5.8), for example, which includes guarded permissions. The corresponding "behaviour" task would be defined in a generic package of the following form :-

```
generic                                                          /7.30/
   with FULL_GUARD (OB : in APPLICATION.OBJECT) return BOOLEAN;
package BOUNDED_BUFFER is

   task type BEHAVIOUR is

      entry INITIALISE (OB : in APPLICATION.OBJECT);

      entry PUT_OPS_REQUEST;
      entry PUT_OPS_COMPLETION;

      entry GET_OPS_REQUEST;
      entry GET_OPS_COMPLETION;

      entry FULL_GUARD_REQUEST;
      entry FULL_GUARD_COMPLETION;

      entry OTHER_OPS_REQUEST;
      entry OTHER_OPS_COMPLETION;
```

```
    end BEHAVIOUR;
end BOUNDED_BUFFER;

package body BOUNDED_BUFFER is
    task body BEHAVIOUR is
        SIZE : NATURAL := 0;
        F_GUARD : BOOLEAN := FALSE;
        SELF : APPLICATION.OBJECT;
        procedure EVALUATE_GUARDS is
        begin
            F_GUARD := FULL_GUARD(SELF);
        end;
    begin
        accept INITIALISE (OB : in APPLICATION.OBJECT) do
            SELF := OB;
        end;
        loop
            select
                when not F_GUARD =>
                    accept PUT_OPS_REQUEST;
            or
                when not SIZE = 0 =>
                    accept GET_OPS_REQUEST;
            or
                accept FULL_GUARD_REQUEST;
            or
                accept OTHER_OPS_REQUEST;
            or
                accept PUT_OPS_COMPLETION do
                    SIZE := SIZE + 1 ;
                    EVALUATE_GUARDS;
                end;
            or
                accept GET_OPS_COMPLETION do
                    SIZE := SIZE - 1 ;
                    EVALUATE_GUARDS;
                end;
            or
                accept FULL_GUARD_COMPLETION;
            or
                accept OTHER_OPS_COMPLETION do
                    EVALUATE_GUARDS;
                end;
            end select ;
        end loop;
    end BEHAVIOUR;
end BOUNDED_BUFFER;
```

Before the required "behaviour" task can be appended onto the state list for an instance of a class such as *GUARDED_QUEUE* (5.9) controlled by *BOUNDED_BUFFER*, this package must first be instantiated with the "implementation" subprogram corresponding to the guard method. *GUARDED_QUEUE*, for example, would be translated into the following package :-

```
with APPLICATION, BOUNDED_BUFFER                                    /7.31/
package GUARDED_QUEUE is
    package QUEUE is new BOUNDED_BUFFER
                                (SIMPLE_QUEUE.IS_FULL => FULL_GUARD);
    function PART_OF (OB : in APPLICATION.OBJECT)
                                                  return QUEUE.BEHAVIOUR;
    function CREATE return APPLICATION.OBJECT;
end;
```

The task at the end of the state list for *GUARDED_QUEUE* is generated from an instance of the generic package *BOUNDED_BUFFER* provided with the guard method *SIMPLE_QUEUE.IS_FULL* as actual parameter. Evaluation of the procedure *EVALUATE_GUARDS* on the completion of each non-guard method will result in the guard variable *F_GUARD* being assigned to the value returned by the subprogram implementing *IS_FULL* as required.

7.7 Executable Objects

According to the DRAGOON distribution model outlined in the previous chapter, in order to execute instances of a user-defined class on a particular type of machine it is first necessary to define the appropriate executable subclass. Submission of an executable class to the DRAGOON compilation systems results not only in the generation of new Ada software, but also the complete translation of the class into a self-contained executable load module. In typical compiler terminology, executable classes are not only translated into Ada modules and then subsequently "compiled", but the "linker" is also invoked to combine the necessary Ada modules into an executable image of the class.

Since Ada recognises only library subprograms as "main programs" that can be mapped into an executable image (i.e. linked), when processing an executable class the DRAGOON compilation system must generate some kind of library subprogram which can be used by the Ada compilation system to generate an executable image. Consider, for example, the version of the *FORWARDER* (5.8) needed in the distributed version of the supermarket system for execution on a Sun Workstation. This class would be defined as an executable class of the following kind :-

```
class SUN_EXECUTABLE_FORWARDER                                    /7.32/
    inherits FORWARDER, SUN_EXECUTION_SUPPORT;
end;
```

where *SUN_EXECUTION_SUPPORT* is an execution support class for Sun workstations. To make it possible for the Ada compilation system to generate an executable image of such a class, the DRAGOON translation tool would generate a subprogram of the following form :-

```
with APPLICATION, FORWARDER;                                      /7.33/
with <other "system" packages >;
procedure FORWARDER_ROOT is
    OB : APPLICATION.OBJECT;
begin
    OB := FORWARDER.CREATE;
end;
```

The only actions this procedure actually performs is to declare an access variable for the associated application object and to instantiate it using the corresponding *CREATE* subprogram. Its most important role however is to act as the "root" of the dependency graph for all the Ada library units which need to be linked together to provide the desired properties. This is not just the Ada library units generated from the application class (i.e. *FORWARDER*) and its ancestors, but also the library units needed to support the properties the executable class inherits from the execution support class. Even in the case of the canonical form, which is the most primitive execution support class for the machine concerned, a certain amount of Ada code will be needed to support the *LOAD_ONTO* method, and the other properties associated with processes in the target environment (i.e. Sun Workstations in this case). In general, however, execution support classes may export an unlimited number of methods representing the operations performable on such processes - methods whose invocation must be supported across the network.

The potential strategies for supporting distributed execution of DRAGOON objects will be outlined in the following section. For the present it is only necessary to be aware that a certain quantity of Ada software must be included in the "main program" for an executable class in order to include the functionality provided by the execution support class. The Ada library units concerned, however, are not application specific, and form part of the predefined DRAGOON environment.

As illustrated by *MC_68000_EXECUTABLE_CHECKOUT_POINT* (6.8) in the previous chapter, it is possible for the user to redefine methods in the body of executable objects, and even to introduce new variables. In such circumstances, an additional package is produced by the translation tool (according to the principles outlined

above) to implement the methods introduced or redefined by the programmer in the executable object, and define a record holding the new variables. The special "root" procedure outlined above, therefore, is a client of this package, rather than of the package corresponding to the inherited application class.

Cross Compilation

One of the most powerful features of execution support classes in the DRAGOON distribution model is that they permit the compilation of objects for different machine types to be modelled in the language, and hence represented in the normal class library. The different machine types that can be supported, however, and the stages to which a class can be translated before specific targeting is necessary, depends largely on the cross compilation facilities provides by the host Ada system.

Many Ada compilation systems supporting compilation to numerous targets use a special intermediate code to provide a machine independent representation of Ada modules. Ada library units accepted by the compilation system are thus stored in this intermediate form. It is only when the linker is used to generate executable modules that dependency on a particular type of machine is fixed and the final part of the translation to machine code is performed. The same intermediate representation of an Ada module can therefore be used for many different machine types without recompilation of the original Ada.

When used with such a compilation system Ada code produced by the translation system can be immediately compiled into the intermediate form. If such a system is not available, however, the Ada can not be "compiled" until the mapping to a particular machine type is fixed (by the definition of an executable object), and an appropriately targeted compiler can be used. In this case, Ada is essentially acting as the intermediate code for DRAGOON. In general, the Ada code produced by the translation tool can be immediately compiled to the "lowest" form maintaining machine independence, which in certain cases may be Ada itself.

7.8 Distribution

The previous sections have described how the "non-distributed" features of DRAGOON are translated into "pure", standard Ada for execution. It is not possible to support the distribution features of DRAGOON entirely in Ada, however, because the language does not have in-built support for distribution. Implementation of the DRAGOON distribution model is bound, to a certain extent, to involve some interaction with the facilities of the supporting environment.

If DRAGOON software is required to execute on a particular type of network operating/communication system, a translation could be devised to arrange for the necessary interaction with (i.e. system calls to) the services of that particular system. In general, however, it is clearly unsatisfactory to have to develop a different translation, and associated tool, for each different type of network on which DRAGOON software is to run, since this entirely defeats the portability advantages of compilation into Ada.

There are two basic approaches for overcoming this problem. One is to *minimise* the dependence of the transformation on the idiosynchrasies of particular operating and communications systems by defining, in the spirit of DIADEM [Atkinson et al, 88], a minimal "standard" set of primitives, which must be supported by any network on which DRAGOON software is to execute. The other approach is, in a sense, to *maximise* the dependence of the translation on the underlying system, by developing a distributed run-time system specifically for the execution of DRAGOON software. A dedicated run-time system of this kind can provide much greater efficiency since it is able to support primitives explicitly tailored to the DRAGOON object model, and remote communication in terms of method calls. A prototype version of a specialised DRAGOON distributed run-time system, known as the "DRAGON Distributed Executive" (DDX) is under developed in the DRAGON project. The price of this efficiency, however, is lower portability, since the work required to implement the dedicated DRAGOON run-time system for different networks is much greater than that required to reimplement the minimal interface on top of the host operating system.

As the details of the specialised DRAGOON run-time system (DDX) are fully described elsewhere [Bayan et. al, 89], the remainder of this chapter will concentrate on the alternative "standard interface" approach.

7.8.1 Supporting Distributed Execution

There are two main problems involved in supporting the execution of objects distributed over a network :-

- object identification,
- remote communication.

For objects in a system to interact and cooperate effectively they clearly require some means of uniquely identifying the objects they wish to communicate with. Once they have a means of identifying each other, a mechanism must be provided by which they may synchronise and communicate. If the services provided by the interface to the transport layer of the network are intended to be the minimum

necessary, most of these requirements must be handled by the Ada code produced by the translation tool, rather than the underlying system.

One project which addressed the problems of supporting remote interactions by automatically generated Ada code interacting with a minimal interface to the underlying communication system was the DIADEM project. This was concerned specifically with supporting the distributed execution of Ada software using the rendezvous protocol. In view of the similarities of this with the method call protocol of DRAGOON, the DIADEM implementation strategy turns out to be highly suitable for DRAGOON also.

To permit remote objects to identify each other in the network, DIADEM introduced the concept of a "remote access type". This is simply a record which stores the local reference to an object alongside the unique network address of the machine on which it resides. Together these uniquely identify executable objects in the system. The DIADEM implementation of remote rendezvous transactions is based on the use of dynamically generated tasks to act as the local surrogates for remote callers. These surrogate tasks are generated by a permanent "port" task which listens at a prearranged communication point for incoming calls.

The fine details of the DIADEM technique for implementing remote communication are explained fully in [Atkinson et al, 88], and there is little value in reiterating them here. A more important question is how this basic strategy can be integrated with the translation techniques outlined above.

Object Identification

In DRAGOON, the only objects which may be translated into executable objects for distribution over the machines of a network are "virtual node" objects conforming to the appropriate construction rules. It is only in the translation of such objects, therefore, that consideration must be given to distribution aspects. In particular, the methods exported by virtual node objects may be called remotely, and their *port* variables may refer to remote objects. Since virtual node objects may also be used like any others as private components in the implementation of larger encapsulating objects, the methods of such objects may also be called locally, and their variables refer to local objects.

The choice between these two possibilities is made when an executable subclass of a virtual node is defined. Submitting such a class to the compilation system indicates that the methods of the class will be called remotely, and that the port variables will refer to remote objects. At the time of *translation* of virtual node classes, therefore, it is not known whether the implementation of methods and port variables will have to support remote or local transactions, and hence both possibilities must be

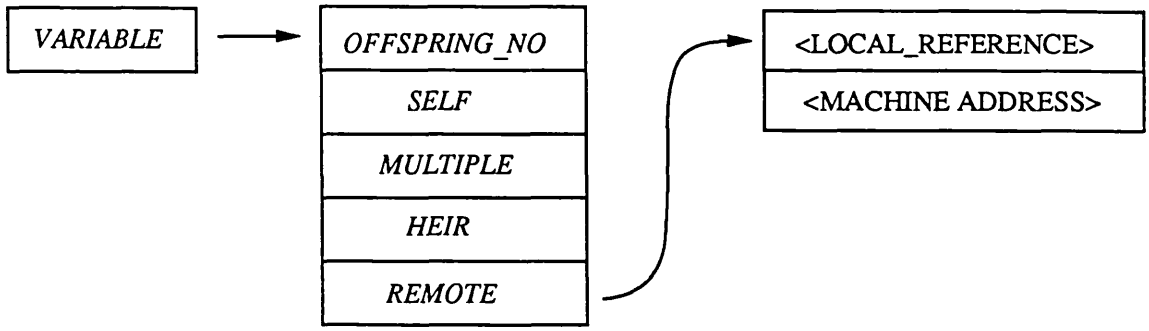


Figure 7.4: Data Structure for Remote Objects.

accommodated in the translation.

To cope with the possibility that the port variables of a virtual node class may refer to remote objects (if inherited by an executable subclass), an extra field is required in the first node of all the state lists - that is, in the record *STATE* in the predefined package *APPLICATION*. Strictly speaking, it is only necessary for the state list of virtual node objects to contain this extra field, since only port variables can refer to such objects. Since it is essential that the first node of *every* object is of the same type, this field, called *REMOTE*, is added to the first node of all state lists.

Like all the other reference fields in this record (e.g. *HEIR*, *MULTIPLE* etc.) this record is arbitrarily defined to be of type *APPLICATION.OBJECT*. In a state list representing a local object (i.e. one in the same executable object) the *REMOTE* field has the value "**null**" and is essentially redundant. Remote objects, however, are represented by data structures of the form illustrated in figure 7.4, in which the *REMOTE* field refers to a record storing the "remote access type" of the object. In this case the other fields of the *APPLICATION.OBJECT* record are unused.

The exact form of the record referenced by the *REMOTE* field is system dependent, but typically will hold the network address of the machine on which the object resides, and the local reference of the object on this machine. Since this record is clearly not of type *APPLICATION.STATE* the generic function *UNCHECKED-_CONVERSION* has to be used as for the other fields to change its apparent type to that required.

The port variables of executable objects are thus implemented as normal Ada access values, but refer to "remote object" structures rather than normal state lists.

Remote Communication

The other aspect of distribution which needs to be accommodated in the translation of virtual node classes is support for remote communication. In the DIADEM scheme this requires the definition of special Ada program units - "port" tasks to listen at prearranged communication points, "surrogate" tasks to issue local calls, dispatching procedures to translate local calls to the appropriate transport layer messages, and special template packages to define the types of the exchanged data structures.

When a virtual node object is translated into Ada, therefore, additional Ada packages are defined containing the necessary program constructs to handle remote and local transactions. These are some of the "additional" packages referred to in the second "**with**" clause in (7.33) above. Clearly they are not included in a program which has local instances of the virtual node class.

As well as the additional packages, however, slight modifications have to be made to the "main" package and selection shell. In addition to the "implementing" subprograms contained in the inner package *SELF*, an additional set of subprograms are defined in another inner package called *REMOTE*. It is the subprograms in this package which are responsible for implementing remote method calls on the caller's side. In other words, when invoked from a selection shell, these subprograms use the object identifier references by the *REMOTE* field to dispatch the appropriate transport layer messages to the remote site. In effect, these subprograms provide a "remote" implementation of the corresponding methods while the subprograms in *SELF* provide a "local" implementation.

Naturally, it is the selection shells corresponding to the methods which are responsible for choosing between these two by checking the value of the remote field of the *APPLICATION.STATE* record referenced by the access variables. It is this fields which is also used to check that at run-time that "ingenerate" references are not exported from executable objects. If one of the subprograms in *REMOTE* is invoked with an "object" parameter whose *REMOTE* field is "**null**, the appropriate exception is raised.

Chapter 8

Conclusion

This thesis has introduced the principal features of the object-oriented language DRAGOON, and by means of their application in a simple case study, the rationale behind them. This concluding chapter briefly outlines other aspects of the DRAGON project and then evaluates the contribution of the language to object-oriented programming and to software engineering in general.

8.1 Other aspects of DRAGON

Although much of the effort in the DRAGON project has focussed on DRAGOON, particularly the development of the preprocessor into Ada [Genolini and Cardigno, 89], and the distributed run-time executive [Bayan et al, 89], the language is actually only one of a number of developments providing a coherent framework for all major aspects of software development [Di Maio et al, 89].

Reuse Toolset

The most crucial part of the design process, and yet the one which is hardest to formalise or define prescriptive rules for, is the very first stage when vague ideas crystalise into a more concrete design. It is at this early stage that the potential for reuse is at its greatest, but also easiest to miss. The project has therefore developed a highly interactive tool, known as the Designer's Notepad (DNP) [Sommerville et al, 1989b] which aims to assist the designer in capturing possible designs, with a view to reusing existing components. It has been described as a form of "electronic paper" replacing the traditional pencil and "envelope".

Naturally, to perform its role, the DNP requires information about the reusable components available to the designer. The Software Components Catalogue (SCC) has been developed to provides a user friendly interface to the components stored in the Component Information Store (CIS), and can be accessed directly by the DNP.

It provides both browsing and retrieval facilities based on the notion of frames which a user "fills in" to describe the properties of the object required.

Design Method

Although it is impossible to define an exhaustive "cookbook" of prescriptive rules for constructing software, *design methods* are nevertheless useful for providing a general framework to guide the development process. This is particularly so when emphasis is placed on guiding the progress of designs according to the contents of the component library.

There are already quite a number of so called "object-oriented" design methods in existence, such as HOOD [87], [Berard, 85] and [Booch, 87], which use the "object" or "class" as the main system building blocks. These, however all take a compositional approach to object design in which new objects are constructed from others by clientship. As pointed out in chapter 4, object-oriented *languages* support another equally powerful reuse mechanism known as inheritance, which permits classes to be generated by expressing their difference to other classes.

The design method developed in the project, known as DEMON (DEsign method for an Object-oriented Notation) [Bott et al, 89] recognises this alternative approach and places equal weight on design using the clientship and inheritance mechanisms. The method first attempts to produce the required "system" class by modifying existing components in the library by inheritance. If this fails, the required class has to be generated from (possibly modified versions of) smaller classes in the library by clientship. The DEMON design process is therefore inherently recursive. At each point in the dependency hierarchy attempts are made to derive the required component by inheritance, as well as by the usual compositional approach.

Formal Aspects

If the many potential advantages of software reuse are to be realised, it must be pursued in a rigorous and controlled fashion. In particular, it is important that reusable components do what they claim to do, and that they are employed in the manner intended by the designer. Formal methods, based on algebraic specification techniques, have been developed in DRAGON to try to ensure that both these criteria are satisfied.

The major effort has centered on the definition of formal semantics for DRAGOON, since this provides a basis for reasoning about program correctness. The approach developed extends the traditional denotational style associated with abstract data types to handle the inheritance and concurrency features of DRAGOON

[Breu and Zucca, 89]. Formal techniques have also been developed for describing the properties of reusable components in the library [Wirsing, 88].

8.2 Contribution of DRAGOON

Together with these accompanying development tools, it is felt that the DRAGOON language makes two important contributions to the technology of large-scale software systems. Because it is intimately allied to Ada, not only being geared to implementation in the language, but adopting its philosophy and conventions, DRAGOON is a highly practical language, "inheriting" Ada's stability and portability advantages. As well as making the powerful reuse mechanisms of object-oriented programming available to the Ada world, therefore, it brings the advantages of the Ada industry to the object-oriented programming world.

Despite this highly practical side, however, the language also introduces some important new ideas for extending object-oriented programming into the the domains of concurrency and distribution. Until now no existing language has fully reconciled the inheritance mechanism with the demands of concurrency and distribution. Languages like Eiffel, which exploit the power of inheritance for reuse and extensibility, do not support concurrency or distribution, while those that do such as POOL and Emerald do not support inheritance. DRAGOON, however, employs variations on the multiple inheritance theme to support the specification of synchronisation constraints and distributability in a manner that is consistent with the normal inheritance mechanism.

The basis of DRAGOON's approach to these problems is the extension of the idea of classes/objects as a unifying concept for supporting as many different programming requirements as possible. Thus, in DRAGOON classes/objects are employed as units of

- modularity,
- information hiding,
- abstraction,
- compilation,
- typing,
- reuse,
- independent execution (i.e. processes),
- synchronisation (c.f. monitors),
- distribution, (i.e. virtual node objects),
- reconfiguration, (i.e. executable objects) ,

- storage (I/O).

Many languages support a variety of these ideas, but often by means of distinct and disjoint constructs. By using objects/classes for all these different purposes DRAGOON encourages crucial design decisions (i.e. is an object to be active? behavioured? distributable? etc.) to be delayed until the latest possible moment in the design process, thereby minimising the work required to alter these decisions in other systems, and thus increasing the reusability of components.

There is a big difference in scope between DRAGOON and other recently developed object-oriented enhancements of Ada, therefore, since DRAGOON addresses a much wider range of applications areas. Moreover, only one of these, Ada++ [Forestier et al, 89], has attempted to *integrate* object-oriented concepts with Ada, rather than *superimpose* them at a higher level. The object-oriented features of Classic Ada [Bach, 89], in contrast, are strongly based on those of Smalltalk, while those of Innovada [Simonian and Crone, 88] closely resembling Flavors (a Lisp based language) [Moon, 86]. The programming paradigms adopted by these languages are thus primarily those of existing object-oriented languages alien Ada.

Having extended the fledgling object-oriented paradigm in the direction of concurrency and distribution, as well as in the direction of Ada, it is hoped that DRAGOON will reach a large number of users, and prove an important addition to the arsenal of software developers. By providing a general overview of the language, and describing the rationale behind its most important and innovative features, it is hoped also that this thesis represents a small contribution to this goal.

References

Agha, G., 1986,
"An Overview of Actor Languages", *SIGPLAN Notices*, vol. 21, no. 10, pp. 58-67.

Almes, G.T., Black, A.P., Lazowska, E.D. and Noe, J.D., 1985,
"The Eden System: A technical review", *IEEE Transactions on Software Engineering*, vol. SE-11, no. 1, pp. 43-59.

Andrews, G.R. and Schneider, F.B., 1983,
"Concepts and Notations for Concurrent Programming", *Computing Surveys*, March, vol. 15, no. 1, pp. 3-43.

Andrews, G.R. and Olson, R., 1986,
"The evolution of the SR programming language", *Distributed Computing*, no. 1, July, pp. 133-149.

America, P., 1987,
"POOL-T: A Parallel Object-Oriented Language", in *Object Oriented Concurrent Programming*, MIT Press,pp.. 199-220.

America, P., 1989,
"Issues in the Design of Parallel Object-Oriented Languages", *Esprit Project 415 Document 452*, Philips Research Laboratories, Eindhoven, March.

Atkinson, C., 1988,
"Programming Distributed Systems in Ada", *Proc. IFIP Conf. on Hardware and Software for Real-Time Process Control*, Warsaw, (ed. Zalewski and Ehrenberger), North-Holland Publishing Company, pp. 45-53.

Atkinson, C., Moreton, T. and Natali, A., 1988,
Ada for Distributed Systems, The Ada Companion Series, Cambridge University Press.

Atkinson, C. and Di Maio, A., 1989,
"From DIADEM to DRAGOON", in *Proc. Distributed Ada Symposium 1989 (DA '89)*, Dec., The Ada Companion Series, Cambridge University Press, pp. 109-139.

Atkinson, C., Goldsack, S.J., Di Maio, A. and Bayan, R., 1990,
"Object-Oriented Concurrency and Distribution in DRAGOON", to appear in *The Journal of Object-Oriented Programming.*

Bach, W.W., 1989,
"Is Ada Really an Object-Oriented Programming Language", *The Journal of Pascal, Ada and Modula-2*, March/April, pp. 19-25.

Banerjee, J., Chou, H., Garza, J.F., Kim, W., Woelk, D., Ballou, N. and Kim, H., 1987,
"Data Model Issues for Object-Oriented Applications", *ACM Transactions on Office Information Systems*, vol. 5, no. 1, pp. 3-26.

Bayan, R., Destombes, C. and Kaag, F., 1989,
"DDX Distributed Executive Interface for MOTOROLA 68020 Multiprocessor DRAGOON applications", *Project Report, DRAGON/WP2.T5/TECSI/1.*

Bayan, R., Crespi-Reghizzi, S. and Di Maio, A., 1989,
"DRAGOON: Distribution and Reconfiguration of Ada Programs" *Workshop on Communication Networks and Distributed Operating Systems within the Space Environment*, ESA ESTEC, Noordwijk, Netherlands, Oct.

Bennet, J.K., 1987,
"The Design and Implementation of Distributed Smalltalk", *OOPSLA '87 Conference Proceedings*, SIGPLAN Notices (Special Issue), vol. 22, no. 12, pp. 318-329.

Berard, E., 1985,
Object-Oriented Design Handbook for Ada Software, EVB Software Engineering Inc.

Birtwistle, G., Dahl, O., Myrhaug, B. and Nygaard, K., 1983,
Simula Begin, Studentliteratur (Lund) and Auerbach Publishers Inc.

Blair, G.S., Gallagher, J.J. and Malik, J., 1989,
"Genericity vs Inheritance vs Delegation vs Conformance vs ...", to appear in *The Journal of Object-Oriented Programming.*

Black, A., Hutchinson, N., Jul. E. and Levy, H., 1986,
"Object Structure in the Emerald System", *OOPSLA '86 Conference Proceedings*, SIGPLAN Notices (Special Issue), vol. 21, no. 11, pp. 78-86.

Black, A., Hutchinson, N., Jul, E., Levy, H. and Carter, L., 1987,
"Distribution and Abstract Types in Emerald", *IEEE Transactions on Software Engineering*, vol. SE13, no. 1, Jan., pp. 65-76.

Boari, M., Crespi-Reghizzi, S., Dapra, A., Maderna, F. and Natali, A., 1984,
"Multi-microprocessor Programming Techniques: MML a New Set of Tools" *IEEE Computer*, Jan., 47-59.

Booch, G., 1983,
Software Engineering with Ada, Benjamin/Cummings Publishing Company.

Booch, G., 1987,
Software Components with Ada, Benjamin/Cummings Publishing Company.

Borning, A., 1986,
"Classes versus Prototypes in Object-Oriented Languages", *Fall Joint Computer Conference*, ACM/IEEE, Dallas, Texas.

Bott, F., Higgs, C. and Ormsby, A., 1989,
"DEMON User Manual: Version 3.0", *Project Report, DRAGON/WP2.T3-/UCW/5.*

Breu, R. and Zucca, E., 1989,
"An Algebraic Compositional Semantics of an Object Notation with Concurrency", *Project Report, DRAGON/WP2.T1/D2.*

Brinch Hansen, P., 1973,
"Concurrent Programming Concepts", *ACM Computing Surveys*, vol. 5, no. 4, 223-245.

Brobow, D.G., Stefik, M.J., 1982,
LOOPS: an Object-Oriented Programming System for Interlsp, Technical Report, Xerox PARC.

Burns, A., Lister, A.M. and Wellings, A.J., 1985,
"A Review of Ada Tasking", *York Computer Science Report No. 78*, Dept. of Computer Science, University of York.

Campbell, R.H. and Habermann, A.N., 1974,
"The Specification of Process Synchronisation by Path Expressions", *Lecture Notes in Computer Science*, vol. 16, Springer-Verlag, pp. 89-102.

Cardelli, L. and Wegner, P., 1985,
"On Understanding Types, Data Abstraction and Polymorphism", *ACM Computing Surveys*, vol. 17, no. 4, pp. 471-522.

Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B. and Nelson, G., 1988,
Modula-3 Report, Systems Research Center, Paolo, Alto, California.

Chikayana, T., 1984, "ESP Reference Manual",
ICOT Technical Report: TR-044 Tokyo, Japan.

Comandos, 1987,
"Comandos Object Oriented Architecture", *Intermediate Report on Global Architecture*, Feb., 1987.

Conway, M.E., 1963,
"A Multiprocessor System Design" *Proc. AFIPS Fall Jt. Computer Conf.*, vol. 24, Spartan Books, Maryland, pp. 139-146.

Cook, W.R., 1989,
"A Proposal for Making Eiffel Type-Safe", in *ECOOP'89 Conference Proceedings*, (ed. Cook, S.), BCS Workshop Series, Cambridge University Press, pp. 57-70.

Cox, B., 1986,
Object-Oriented Programming - An Evolutionary Approach, Addison-Wesley Publishing Company.

Crowl, L.A., 1988,
"A Uniform Object Model for Parallel Programming", position paper for *Proc. ACM Sigplan Workshop on Object-Based Concurrent Programming*, Sept. 88, SIGPLAN Notices, vol. 24, no. 4, pp. 79-80.

Davison, A., 1989,
Polka: A Parlog Object-Oriented Language, PhD Thesis, Department of Computing, Imperial College, London.

Dasgupta, P., 1986,
"A Probe Based Monitoring Scheme for an Object-Oriented Distributed Operating System", *OOPSLA'86 Conference Proceedings*, SIGPLAN Notices (Special Issue), vol. 21, no. 11, pp. 57-66.

Dijkstra, E.W., 1968,
"Cooperating Sequential Processes", in *Programming Languages*, (ed. F. Genuys), Academic Press.

Di Maio, A., Bott, F., Sommerville, I., Bayan, R. and Wirsing, M., 1989,
"The DRAGON Project", *1989 Esprit Conference*, Brussels, Nov/Dec, pp. 554-567.

Di Maio, A., Cardigno, C., Genolini, S., Crespi-Reghizzi, S., Bayan, R., Destombes, C., Atkinson, C. and Goldsack S.J., 1988,
"DRAGOON: The language and its Implementation", *Project Report DRAGON/WP1.T6/TXT/11.*

DoD, 1983,
Reference Manual for the Ada Programming Language, Ada Joint Program Office, Department of Defence, ANSI/MIL-STD-1815A.

Dulay, N., Kramer, J., Magee, J., Sloman, S. and Twidle, K., 1987,
"Distributed System Construction: Experience with the CONIC Toolkit", *Experiences with Distributed Systems: Proc. Int. Workshop*, Kaiserslautern, FRG, pp. 189-212.

Forestier, J.P., Forarino, C. and Franchi-Zannettacci, P., 1989,
"Ada++: A Class and Inheritance Extension For Ada", *Proc. of the Ada Europe Int. Conf.*, Madrid, June, pp. 189-212.

Gautier, R.J., Bott, F. and Elliott, A., 1989,
"Ada Reuse Guidelines", in *Software Reuse with Ada*, (ed. Gautier, R.J. and Wallis, P.J.L.), Peter Peregrinus Ltd.

Genolini, S., Di Maio, A., Cardigno, C., Goldsack, S. and Atkinson, C., 1989,
"Specifying Synchronisation Constraints in a Concurrent Object-Oriented Language", *Proc. First Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS'89)*, Paris - La Defense, Nov.

Genolini, S. and Cardigno, C., 1989,
"Detailed Design of the DRAGOON Compilation System", *Project Report, DRAGON/WP2.T4/TXT/6.*

Goldsack, S.J., Atkinson, C., Natali, A., Di Maio, A., Maderna, F. and Moreton, T., 1987,
"Ada for Distributed Systems: A Library of Virtual Nodes", *Ada Components: Libraries and Tools, Proc. Ada-Europe Int. Conf.*, The Ada Companion Series, Cambridge University Press, pp. 253-265.

Goldsack, S.J., 1989,
Specifying Requirements: An Introduction to the FOREST Approach, Imperial College Research Report, Dept. of Computing, London SW7 2BZ.

Goldsack, S.J. and Atkinson, C., 1989,
"An Object-Oriented Approach to virtual Nodes: Are Package Types an Answer?", *Proc. 3rd Int. Workshop on Real-Time Ada Issues*, Nemacolin Woodlands, Farmington, PA, June, pp. 95-101.

Goldberg, A. and Robson, D., 1983,
Smalltalk-80: The Language and its Implementation, Addison-Wesley.

Gregory, S., 1987,
Parallel Logic Programming in PARLOG, Addison-Wesley Publishing Company.

Hoare, C.A.R., 1974,
"Monitors: An Operating System Structuring Concept", *Communications of the ACM*, vol. 17, no. 10, pp. 549-557.

Hoare, C.A.R., 1978,
"Communicating Sequential Processes", *Communications of the ACM*, vol. 21, no. 8, pp. 666-677.

HOOD., 1987,
The HOOD Manual, Issue 2.1.

Hutcheon, A.D. and Wellings A.J., 1987,
"Ada for Distributed Systems", *Computer Standards and Interfaces*, vol. 6, no. 1, pp. 71-82.

Hutcheon, A.D. and Wellings A.J., 1988,
"The Virtual Node Approach to Designing Distributed Ada Programs", *Proc. 7th Ada UK International Conference*, Ada User, vol. 9, (Supplement), pp. 35-42.

Ichbiah, J.D, Barnes, J.G.P., Firth, R.J. and Woodger, M., 1986,
Rationale for the Design of the Ada Programming Language, Honeywell Systems and Research Center, Minneapolis, and ALSYS, France.

Ishikawa, Y. and Tokoro, M., 1987,
"Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation", in *Object-Oriented Concurrent Programming*, (ed. Yonezawa and Tokoro), MIT Press, pp. 159-198.

Jackson, M.A., 1975,
Principals of Program Design, Academic Press.

Kafura, D.G. 88,
Concurrent Object-Oriented Real-Time Systems, Technical Report, TR 88-47, Dept. of Computer Science, Virginia Tech.

Kafura, D.G. and Lee, K.H., 89,
"Inheritance in Actor Based Concurrent Object-Oriented Languages", *ECOOP'89 Conference Proceedings*, (ed. Cook, S.), BCS Workshop Series, Cambridge University Press, pp. 131-145.

Kaiser, G.E., Popovich, S.S., Hseush, W. and Wu, S.F., 1989,
"MELDing Multiple Granularities of Parallelism",
ECOOP'89 Conference Proceedings, (ed. Cook, S.), BCS Workshop Series, Cambridge University Press, pp. 147-166.

Khosla, S., 1988,
System Specification: A Deontic Approach, PhD Thesis, Dept. of Computing. Imperial College, London.

Kristensen, B.B, Madsen, O.L., Moller-Pederson, B. and Nygaard, K., 1985,
"Multisequential Execution for the BETA Programming Language", *SIGPLAN Notices*, Apr., vol. 20, no. 4, pp. 57-70.

Kramer, J. and Magee, J., 1985,
"Dynamic Configuration for Distributed Systems", *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 424-435.

Kramer, J. and Magee, J., 1988,
"A Model for Change Management", *IEEE Workshop on Future Trends in Distributed Computing Systems in the 1990s*, Hong Kong, Sept., pp. 286-295.

Lieberman, H., 1986,
"Using Prototypical Objects to Implement Shared Behaviour in Object--Oriented Systems", *OOPSLA'86 Proceedings*, SIGPLAN Notices, vol. 21, no. 11, pp. 214-223.

Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J.C., Scheifler, R. and Snyder, A., 1981,
CLU Reference Manual, Springer-Verlag.

Liskov, B., 1982,
"On Linguistic Support for Distributed Programs", *IEEE Transactions*, vol. SE-8, no. 3, pp. 203-210.

McIlroy, M.D., 1969,
"Mass Produced Software Components", in *Software Engineering*, report on a conference sponsored by the NATO Science Committee, Garmisch, W. Germany, 1968, pp. 138-150.

MacLennan, B.J., 1982,
"Values and Objects in Programming Languages", *SIGPLAN Notices*, vol. 17, no. 12, pp. 70-79.

May, D. and Shepherd, R., 1985,
"Occam and the Transputer", in *Concurrent Languages in Distributed Systems*, North-Holland.

Matsumoto, Y., 1984,
"Some Experiences in Promoting Reusable Software Presentation in Higher Abstract Level", *IEEE Transactions on Software Engineering*, vol SE-10, no. 5, pp. 502-513.

Meyer, B., 1986,
"Genericity versus Inheritance", *OOPSLA'86 Proceedings*, SIGPLAN Notices, vol. 21, no. 11, pp. 391-405.

Meyer, B., 1988,
Object-Oriented Software Construction, Prentice Hall, 1988.

Meyer, B. and Nerson, J., 1989,
Eiffel Release 2.2 Overview, Interactive Software Engineering Inc., 1989.

Milner, R., 1984,
"A Proposal for Standard ML", *Proc. of the Symp. on LISP and Functional programming*, ACM, pp. 184-197.

Moon, D.A., 1986,
"Object-Oriented Programming with Flavors",
OOPSLA'86 Proceedings, SIGPLAN Notices, vol. 21, no. 11, pp. 1-8.

Mullery, G., 1976,
"CORE - A Method for Controlled Requirement Specification", *Proc. 4th Int. Conf. Software Engineering*, Munich, IEEE Computer Society Press.

Nierstraz, O.M., 1987,
"Active Objects in Hybrid", *OOPSLA'87 Conference Proceedings*, SIGPLAN Notices (Special Issue), vol. 22, no. 12, pp. 243-253.

Ormsby, A., 1990,
"Object-Oriented Design Methods", chapter to appear in *Object-Oriented Languages, Systems and Applications*, (ed. Blair, G., Gallagher, J., Hutchinson, D. and Shepherd, D.), Pittman (in preparation), - ISBN 0-273-03132-5.

Perez, E. P., 1988,
"Simulating Inheritance with Ada", *Ada Letters*, vol. VIII, no. 5, Sept/Oct., pp. 37-46.

Peterson, G. L., 1981,
"Myths about the Mutual Exclusion Problem", *Information Processing Letters*, vol 12, no. 3., pp. 115-126.

Potts, C., Finkelstein, A., Aslett, M. and Booth, J., 1986,
"Structured Common Sense: A Requirements Elicitation and Formalisation Method for Modal Action Logic", *FOREST Deliverable Report 2.*

Schaffert, C., Cooper, T., Bullis. B. and Killian, M., 1986,
"An Introduction to Trellis/Owl", *OOPSLA'86 Conference Proceedings*, SIGPLAN Notices (Special Issue), vol. 21, no. 11, pp. 9-16.

Seidewitz, E., 1987,
"Object-Oriented Programming in Smalltalk and Ada", *OOPSLA'87 Conference Proceedings*, SIGPLAN Notices (Special Issue), vol. 22, no. 12, pp. 202-213.

Shepherd, D., Coote, S., Gallagher, J., Lea, R., Mariani, J. and Scott, A., 1988
"An Object-Oriented Approach to the High Level Programming of Distributed Process Control Applications", *Proc. 3rd Int. Conf. on Computer-Aided Production Engineering*, Michigan, June, pp. 235-342.

Simonian, R. and Crone, M., 1988
"Innovada: True Object-Oriented Programming in Ada", *Journal of Object-Oriented Programming*, Nov./Dec., pp. 14-21.

Sloman, M. and Kramer, J., 1987
Distributed Systems and Computer Networks, Prentice/Hall International.

Sommerville, I., Mariani, J., Haddley, N. and Thomson, R., 1989a
"Software Design With Reuse", Internal Report, Dept. of Computing, Lancaster University, Bailrigg, Lancaster.

Sommerville, I., Haddley, N., Mariani, J. and Thomson, R., 1989b
"The Designer's Notepad - A Hypertext System Tailored for Design", *Proc. Hypertext II Conference*, York, June.

Snodgrass, R., 1983
"An Object-Oriented Command Language", *IEEE Transactions on Software Engineering*, Jan., vol. SE-9, no. 1, pp. 1-8.

Snyder, A., 1986
"Encapsulation and Inheritance", *OOPSLA'87 Conference Proceedings*, SIGPLAN Notices (Special Issue), vol. 22, no. 12, pp. 38-45.

Stroustrup, B., 1986,
The C++ Programming Language, Addison-Wesley.

Tedd, M., Crespi-Reghizzi, S. and Natali, 1984,
Ada for Multi-microprocessors, The Ada Companion Series, Cambridge University Press.

Wegner, P., 1987,
"Dimensions of Object-Based Language Design", *OOPSLA'87 Conference Proceedings*, SIGPLAN Notices (Special Issue), vol. 22, no. 12, pp. 168-182

Wegner, P. and Zdonik, S.B., 1988,
"Inheritance as an Incremental Modification Mechanism or What Like is and Isn't Like", *Proc. ECOOP'88*, Lecture Notes in Computer Science 322, Springer-Verlag, pp. 55-77.

Wirsing, M., 1988,
"Algebraic Description of Reusable Software Components", *Proc. of COMPEURO '88*, Apr., (ed. Milgrom, E. and Wodon, P.), IEEE Computer Society Press, pp. 300-312.

Wirth, N., 1982,
Programming in Modula-2, Springer-Verlag, New York.

von Wright, G.H., 1951,
An Essay in Modal Logic , North-Holland Publishing Company, Amsterdam.

von Wright, G.H., 1980,
"Problems and Prospects of Deontic Logic: A survey", in *Modern Logic - A Survey: Historical, Philosophical and Mathematical Aspects of Modern Logic and its Applications*, (ed. E. Agazzi) D. Reidel Publishing Company, pp. 399-423.

Vishnubhotta, P., 1988,
"Synchronisation and Scheduling in ALPS Objects", *Proc. 8th International Conf. on Distributed Computer Systems*, San Jose, CA, June, IEEE Computer Society Press, pp. 256-264.

Yokote, Y. and Tokoro, M., 1986,
"The Design and Implementation of Concurrent Smalltalk", *OOPSLA'86 Proceedings*, SIGPLAN Notices, vol. 21, no. 11, pp. 331-340.

Yonezawa, Y., Briot, J. and Shibayama, E., 1986,
"Object-Oriented Concurrent programming in ABCL/1", *OOPSLA'86 Proceedings*, SIGPLAN Notices, vol. 21, no. 11, pp. 258-268.