Data Abstraction and Program Design

08-

Data Abstraction and Program Design

Rod Ellis

Principal Lecturer and Head: Division of Software Engineering
School of Computer Science and Information System Engineering,
Polytechnic of Central London

Pitman

PITMAN PUBLISHING
128 Long Acre, London WC2E 9AN

A Division of Longman Group UK Limited

First published in Great Britain 1991

British Library Cataloguing in Publication Data
Ellis, Rod
Data abstraction and program design.
I. Title
005.1

ISBN 0-273-03257-7

Reproduced and printed by photolithography
in Great Britain by Biddles Ltd, Guildford

Preface

The term *Object-based* is used to denote a common conceptual region shared by modern 'conventional' languages, such as MODULA-2 and Ada, and the Object-Oriented languages Smalltalk, C++, Eiffel etc. This region not only encompasses specifically linguistic concepts, of which data abstraction possesses a significance suggested by its prominence in the title of this book, but also an approach to software design without which these concepts lose much of their significance. In other words, these languages are not neutral in respect of the methods of software design with which they are utilised. Their features can be understood, and exploited successfully, only in the light of the design philosophy, in its various versions, that informed their development.

This book provides an introduction to object-based techniques of software construction. Its intended readership falls into two categories:

1. Software Engineering students in higher education who have completed their initial programming courses, probably in a language such as Pascal. The text introduces them to the concepts of 'programming in the large' and develops these along an Object-based trajectory. As such the book will form a useful basis for an academic course suitable for the second year of a software engineering degree programme.

2. Experienced software engineers who have come across references to Object-based techniques in the computer press, or by word of mouth, and who wish to update their knowledge in this area. The treatment in relation to this readership should enable the acquisition of a useful level of competence in some or all of the techniques dealt with, in conjunction with further reading selected from the Bibliographic Notes.

In writing a book with this objective two possible alternative approaches suggest themselves. The first, more radical approach, assumes no prior knowledge on the part of the reader and discusses Object-based concepts without reference to other, more familiar techniques. The second accepts that very few readers are likely to approach the subject with a mind innocent of programming experience and knowledge, and accordingly attempts to modify the 'mind-set' of its intended readership, by working from the reasonably familiar to the unknown in a gradual development.

This latter approach has been adopted for this book, in recognition of its intended readership. Thus the issues are developed through a progression from a Pascal context, through recognisable developments of Pascal, to the fairly remote areas of algebraic specification and Object-Oriented programming languages and techniques. The central, unifying theme of data abstraction is particularly useful in this context in providing a bridge from conventional to Object-based techniques.

Part of the motivation underlying its writing was the recognition of the difficulty that the average student (and the above-average student for that matter) has in coming to terms with the concept of 'programming in the large', or 'architectural' software design. This difficulty, which is endemic to the technology, is often exacerbated by the typical programming methodology course which, with its emphasis on 'programming in the small', or the detailed implementation of algorithms, provides a set of skills largely different from that involved in high-level, structural design. This emphasis induces a fascination with imperative detail that is extremely difficult to displace. The function of Chapter 1 is, accordingly, to get the reader to think in terms of 'programming in the large', or high-level design, to encourage an appreciation of the importance of defining software components in terms of their behaviour rather than their implementation, and to view success in this high-level design activity in terms of its outcome in supporting apparently mundane qualities such as modifiability and maintainability.

Readers who are experienced in software production may well find that Chapter 1 is preaching to the converted, and prefer to skip to Chapter 2. This interprets the criteria of good design identified in Chapter 1 in a more specifically software orientated context, articulated in terms of the vocabulary of modular software design. These ideas are not new, although their recognition in many works on software design seems fairly scanty.

Chapter 3 considers the support provided for the achievement of good modular design by conventional programming languages, for which Pascal is taken as the model. An examination of the deficiencies that this discussion reveals leads to an appreciation of the need for a new program structure — the data abstraction — the ubiquity of which, to program design, forms a core theme of the book.

Chapters 4 and 5 continue the discussion of language design issues by introducing two descendants of Pascal, MODULA-2 and Ada, from the particular standpoint of their support for the data abstraction. The aim of these Chapters is to convey the essential features of these languages whilst, particularly

in the case of Ada, avoiding becoming enmeshed in detailed complexity.

The first five Chapters approach software design from an analytic viewpoint — *what* constitutes good design and how it can be reinforced by linguistic features. The next two Chapters consider design as a synthetic issue — *how* design is carried out as an activity so as to achieve the goals defined earlier, using the tool of the data abstraction.

Chapter 6 is a case study based on a well-known paper by Parnas, showing how the criteria introduced in earlier Chapters can be used to evaluate alternative approaches to design; and how information hiding, a quality closely related to the use of data abstraction, can be 'designed into' a system. Chapter 7 develops this idea but from a different standpoint — a consideration of the application and implementation domains — leading to an exposition of Object-Oriented Design.

A subsidiary theme is developed strongly in the second half of the book: software reuse, and its implications for design. The qualities of integrity and encapsulation conferred by the data abstraction provide the necessary syntactical support for reusable software components, but are not in themselves sufficient to ensure reusability. The nature of the requirements for reuse are discussed in Chapter 8 in a mainly Ada context, but leading to ideas going beyond Ada.

Chapter 8 can be seen as a bridge to two different facets of reuse. The first arises from the obligation of the writer or supplier of a software component to define clearly the semantics of its use. It is precisely those qualities of information hiding, of separation of interface from implementation detail, that make the communication of the semantics of these interfaces a critical issue — it is neither desirable nor, often, possible for the user of such a component to determine its correct usage by inspecting the code that realises its implementation. In Chapter 9 the necessity for formal, or mathematical, semantic specification techniques is discussed, as a precursor to an introduction to a particular formal technique — Algebraic Specification — which has found favour as the most appropriate for specifying components based on the data abstraction.

The second of these facets involves the deployment of a new programming paradigm — Object-Oriented Programming — which combines data abstraction and reuse in a way that both contrasts with and parallels the MODULA-2/Ada approach. Chapter 10 presents the major features of the Object-Oriented paradigm while Chapter 11 contains a comparison of the

characteristics of a representative selection of Object-Oriented programming languages.

Finally, Chapter 12 reviews the contribution made by the data abstraction to modern approaches to software design and considers the tendency for convergence between the two models of exploitation within which the concept is found — a tendency that suggests a considerable unexploited potential.

Acknowledgements

The author would like to thank his colleagues for their forbearance during the gestation of this book, and particularly Mark Priestley for comments on Chapter 9. Thanks are also due for the helpful suggestions of the reviewers of the draft, particularly Gordon Blair of the University of Lancaster, and to John Cushion of Pitman for his patience and convivial encouragement.

Contents

Chapter 1

The Design of Large Systems

1.1 Large Systems and Complexity

In this book we are concerned with the design of large software systems. There are many ways of defining what 'large' means in this context. To use the conventional, if probably the least satisfactory, yardstick, a large software system comprises something of the order of several hundred thousand to a million lines of source code. The unsatisfactory nature of this definition arises from both the vagueness — what exactly is a 'line of source code' — and also the slightly dated nature, of the terminology. Programmers are more used to thinking in terms of statements after all. It does convey, however, an impression of the order of magnitude of the physical manifestation of such a system as a very large body of text.

It may also be objected that the 'real' system is the machine-readable and executable object code version; but the design and implementation of software is a *human activity*, which results finally in a human-readable text — the source program. The generation of the executable form is, or should be, an automated process that requires no involvement on the part of the designer or implementor. So, with all its vagueness, we accept the definition above with its emphasis on the size, and therefore complexity, of the source text.

Even a slight experience of programming will have convinced the reader of the fact that the ease of understanding of a program or program fragment is

adversely affected, in an all too dramatic way, by its size. Moreover, this effect is noticeable in numbers of statements counted in tens, rather than tens of thousands, with an all important threshold occurring, as noted by Brooks in *The Mythical Man-Month*, after *one page*. The difficulty involved in comprehending several hundred thousand statements, and correspondingly, in guiding the generation of several hundred thousand statements so as to realise a design, can easily be imagined therefore. So the question arises, how can the all too finite human mind span this kind of complexity? The answer is not a new one: it is by the use of *abstraction* — in other words, by the removal of inessential detail, by the removal of the trees obscuring the wood.

The key to abstraction lies in the ability to see a system as being composed of a small number of parts or components, each of which can be treated as a simple unit, the inner details of which need not be considered when viewing the whole system.

1.2 Abstraction and Design

The technique of abstraction is well-established in other, more mature branches of engineering. An aircraft wing, for example, is a highly complex structure. When viewed in the context of the predicted performance of the aircraft at the design stage, however, this complexity can be reduced to a few quantitative values: two constants that enable the calculation of the lift and drag of the wing over the range of speeds for which the aircraft is designed, and its weight and major dimensions, particularly including its volume (which normally determines the fuel capacity). Provided the designer of the wing manages, in 'fleshing out' its detailed design, to remain within the framework defined by these few values then the validity of the original predictions will be preserved. Another way of saying this is that, at the *level of abstraction* appropriate to performance calculations, the wing can be considered to *be* this rather small collection of values. Questions as to the airfoil section of the wing, its construction, control surfaces and so on are irrelevant *in this context* and need not be considered at the initial, high-level stages of design.

Abstraction is even more strongly established in electronic engineering with a theoretical basis that allows for any arbitrarily complex system component — an amplifier, say — to be replaced, for the purposes of the analysis of the complete system, by an *equivalent circuit*. An equivalent circuit consists

of one each of the primitive circuit components resistance, capacitance and inductance, together possibly with a current or potential source. All specific details of the component in question are *abstracted away* as far as the rest of the system is concerned.

There are two notions involved in these examples of abstraction:

- the splitting up, or *partitioning*, of the design into discrete parts or components
- the ability to treat these components individually in terms of their effects on the rest of the system, whilst ignoring their internal structure. In these examples these effects can be captured by a small set of values.

1.2.1 Partitioning

When we consider design in these other areas of engineering the process of partitioning is a very natural one. An aircraft or a car can naturally be seen as a collection of components — integrated in the sense that the components fit and work together so as to enable the machine to achieve its designers' objectives. Moreover, the major components of a car such as the engine, the body, the transmission and so on, can be treated as single, monolithic objects at a high level of abstraction, or they may alternatively be viewed as being themselves composed of components populating a lower level of abstraction. For example, the engine is composed of the cylinder block, crankcase, crankshaft, connecting rods and pistons, cylinder head and valve gear, and so on. By choosing an appropriate level of abstraction we can reduce the complexity of the unstructured mass of basic, in the sense of 'having no components' components, that actually result from dismantling the car until no further dismantling is possible.

The process of design naturally follows the path from a high level of abstraction to lower levels — the car designer initially considers very high level 'broad-brush' factors such as the size and major features of the body, the position of the engine and so on. Once these major, high-level features have been determined then the filling out of the next level of detail may be undertaken. Then this process may be continued to successively lower levels until (literally) the nuts and bolts level is reached.

Common sense suggests that there is something wrong if the designer commences the task of designing a car by a detailed analysis of the instrument

panel or the radiator grill. Not only does this offend against commonly held views on 'getting bogged down in detail' but also against the idea that over attention to one part of the design may well lead to an unbalanced final result.

Conventional engineering design techniques, then, suggest that success in the comprehension or design of large or complex systems is very dependent on the ability to consider these systems as comprised of components at various levels of abstraction.

1.3 Partitioning in Software Systems

Turning back to the consideration of software systems, with which we are concerned, it is clear that an important distinction between software and other forms of engineering is that the partitioning of a system into components is far less obvious. An aircraft, for example, is very clearly an assembly of sharply-distinguished components: the wings, tail, fuselage, undercarriage, engines etc. which have obvious and well-defined functions. The lack of any of these components is likely to result in an invalid 'system' — one that does not 'work'. With few exceptions that are only slight deviations from this pattern — and, even then, represent the extreme fringes of innovation ('flying wings' and so on) — this set of components forms the structural model within which the designers of *any* aircraft exercise their ingenuity.

By contrast, there are very few 'standard structural models' to be found generally in software systems, particularly at a high level of abstraction. It is true that many batch systems exhibit an input and an output component, but the rest is both application specific and also dependent on the inspiration of the production team involved. Perhaps more significant is the fact that in the one example where there *is* a generally-recognised standard partitioning, namely compilers, which are generally comprised of components that perform *lexical analysis, syntax analysis, semantic analysis* and *code generation*, there is considerable doubt as to whether this partitioning is a good one.

The software designer, then, does not work within the framework of a high-level structure of standard components. In the typical case a software system might be realised by *many different alternative* sets of components, any one of which would 'work', and it is the task of high-level design to determine