Cliff B. Jones

Software Development

A Rigorous Approach

PRENTICE-HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE

C. A. R. HOARE SERIES EDITOR

SOFTWARE DEVELOPMENT: A RIGOROUS APPROACH

CLIFF B. JONES

IBM ESRI
La Hulpe, Belgium

ENGLEWOOD CLIFFS, N.J. LONDON NEW DELHI SINGAPORE
SYDNEY TOKYO TORONTO WELLINGTON

Library of Congress Cataloguing in Publication Data

JONES, CLIFFORD B 1944–
Software development.
Bibliography: p.
Includes index.
1. Electronic digital computers – Programming.
I. Title
QA76.6.J66 001.6'42 79-14806
ISBN 0-13-821884-6

British Library Cataloguing in Publication Data

JONES, CLIFFORD
Software development.
1. Programming (Electronic computers)
I. Title
001.6'425 QA76.6
ISBN 0-13-821884-6

ISBN 0-13-821884-6

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Dehli*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE., LTD., *Singapore*
PRENTICE-HALL INC., *Englewood Cliffs, New Jersey*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

1 2 3 4 5 84 83 82 81 80

Printed and bound in Great Britain by
A. Wheaton & Co. Ltd, Exeter

TABLE OF SYMBOLS

Arithmetic and Logical Operators

*+ plus, - minus, * multiply, / integer division, ** exponentiation*
$\wedge$ and, $\vee$ or, $\Rightarrow$ implies, $\Leftrightarrow$ equivalence, $\sim$ not
A *for all,* **E** *there exists,* **E!** *there exists exactly one*
ι the unique object
*(**A**$x \epsilon X$)(...) bounded quantification*

Set Notation

{ } set brackets
{x | p(x)} set of elements such that
$x \epsilon X$ is a member
$\cup$ union, $\cap$ intersection, - difference, **union** *distributed union*
$\subseteq$ subset, $\subset$ proper subset
card *cardinality*
Bool = {**TRUE**,**FALSE**}
Nat = {1,2,...}
Nat0 = {0,1,2,...}
Int = {...,-2,-1,0,1,2,...}

List Notation

< > list brackets
hd *head,* **tl** *tail,* **len** *length, (i) indexing,*
| | concatenation, **elems** *collect to a set,* **conc** *distributed concatenation*

Mapping Notation

[ ] mapping brackets
dom *domain,* **rng** *range, (i) application, † overwrite, | restrict*

PREFACE

Without the ability to record its results, neither a science nor even a civilization could make progress. This book teaches a method for recording specifications and designs of computer systems.

Two major problems exist with the production of computer programs (i.e. software). Firstly the created programs are frequently not satisfactory to the people who have to use them and secondly their production is too costly. Newspaper stories of the effects of computer errors are myriad. Programmers are only too aware of the human contribution to these errors. The problem of the productivity of the development of computer systems is also linked to errors. Errors which are made in the specification or early design stages are frequently uncovered late in the development cycle and result in enormous correction costs. Even this does not express the full dissatisfaction with computer programs. There is another major problem in that the systems created are often unnecessarily difficult to use.

The computer industry faces a crisis which has been created by its success. More and more powerful systems are demanded as industry puts greater reliance on computers. In order to be able to produce such systems, new development methods must be employed.

This book is intended for a course which will bring the results of computer science into software development practice. The pre-requisites are simple. Some programming experience is necessary. Furthermore, the reader is assumed to have made errors in his programs and to be dissatisfied with this state of affairs. The programs presented in this book are written in the PL/I language. But, for the majority of the examples, the language constructs which are used are common to nearly all languages. No previous exposure to formal methods is assumed. The book is self-contained in that all required notation is taught.

The ideas covered in this book can be considered under the headings of specification and design. The techniques for recording specifications in a precise and concise way have enabled the author and his colleagues to analyze many existing computing concepts and systems. Not only has this proved to be a powerful way of understanding such systems, but it

has also facilitated their documentation. However, the ultimate advantage of such a specification tool is for the design of new systems. If used in this way, systems will be created whose architecture makes them much more usable.

For design purposes, a method is taught which permits the coherent documentation of a design. A record of why each stage of design is believed to be correct is an integral part of the documentation. For this reason the design documentation can be reviewed easily and the danger of errors remaining undetected is almost eliminated.

Webster's Dictionary defines ***rigorous*** as "scrupulously accurate, precise." The approach taken here is rigorous: it is intended to be precise without being completely formal. Thus the aim is to show ways in which the confidence placed in newly developed software can be drastically increased. Just as in other engineering disciplines, in order to work reliably, one must first learn a formal basis. The final rigorous method is not, however, very mathematical. The crucial point is that, having learnt the theory, one can safely reason at a rigorous level. What one writes will fit into a framework and because of this the developer, or others, will know how to complete the formal details if necessary.

The book is divided into three parts. The first part is devoted to programs which manipulate numbers. Key techniques are introduced based on this simple data type. Pre- and post-conditions provide precise specifications of a program. The development of the control structure of a program can be put on a firm footing by showing how one can prove that the specification is met.

The second part of the book concentrates on data structures. Here, the advantages of documenting programs by using abstract data types are shown to be precision, conciseness, and manipulability. This material shows the applicability of the computer science results to data processing applications. The refinement of such data types onto those available in programming languages is also covered.

The techniques presented in the first two parts of the book are collected in the third part into a coherent development method. A number of examples of the application of this method are given.

Many exercises are included as well as a glossary and appendices which are to be used as reference material both during and after studying the text.

The material has been developed in an industrial environment where it has been taught many times. One pattern in which it has been used is in two or three week intensive courses. In industry courses, the sections marked with * are normally omitted. If all of the remaining material is taught, the students should be able to document both specifications and design in a precise notation. Another possibility is to concentrate on specification methods: in this case chapters 5, 6, 11, and the refinement sections of chapters 12, 14 should also be omitted.

A number of sections have been included (marked *) which make the material suitable for an M.Sc. course. For such an audience, references are provided to related literature.

It is a pleasure to be able to acknowledge the help that I have received with the creation of this book. The current text has evolved over two years in the courses which I taught at the IBM European Systems Research Institute—the students there have provided much useful criticism. In particular Soren Brandt and Andre Fischer checked the whole text.

The method of production of this book deserves fuller description elsewhere. I should, however, like to express my gratitude to Derek Andrews, Neal Eisenberg, and Charles Goldfarb without whom it would have been impossible for me to employ the new technology which was used to typeset this book.

My cooperation with Prentice-Hall International has been a pleasure from beginning to end—I should like to thank Derek Coleman, Ron Decent, Henry Hirschberg and Tony Hoare for their help and encouragement.

A debt of another kind is that to the sources of the ideas. Much of the work presented in this book has been developed in conjunction with my colleagues and friends at the IBM Laboratory Vienna. A continued source of inspiration and criticism has been the meetings of IFIP Working Group 2.3.

Examples and exercises have been taken from the works of P.Henderson, G.Hay, B.Jousset, D.Parnas and P.D.Wright. Permission to use the cartoon in chapter 1 was granted by the A.L.I. Press Agency, Brussels.

CONTENTS

Chapter 1 INTRODUCTION 1
Structure of the Book 1
Background 3
Summary 14
Summary Exercises 14

Part A PROGRAMS WHICH MANIPULATE NUMBERS 17

Chapter 2 SPECIFYING FUNCTIONS 19
Functions 22
Implicit Specifications 23
Propositional Operators 27
Quantifiers 33
Summary 37
Summary Exercises 37

Chapter 3 PROOFS ABOUT FUNCTIONS 38
Unconditional Functions 39
Conditional Functions 43
Recursive Functions 48
*More on Logic 57
Summary 61
Summary Exercises 61

Chapter 4 SPECIFYING PROGRAMS 62
States 63
Operations 65
Specifying Operations 66
General Operations 69
Summary 72
Summary Exercises 72

Chapter 5 PROOFS IN PROGRAM DEVELOPMENT 74
Sequential Statements 81
Conditional Statements 89
Iterative Statements 94

*More on Iterative Statements 101
Summary 112
Summary Exercises 114

Chapter 6 OTHER ISSUES 115
Documenting Algorithms 115
*Predicate Transformers 119

Part B DATA TYPES IN PROGRAM DEVELOPMENT 123

Chapter 7 ON DATA TYPES 125
Arrays 128
Extensions and New Data Types 132
Models of Data Types 134
Summary 135

Chapter 8 SET NOTATION 136
Notation 138
*Inductive Proofs on Sets 144
Use in Specifications 146
Recording Equivalence Relations 149
*Predicates and Sets 154
Summary 155
Summary Exercises 156

Chapter 9 LIST NOTATION 157
Notation 157
*Inductive Proofs on Lists 162
Use in Specifications 164
Summary 171
Summary Exercises 171

Chapter 10 DATA TYPE INVARIANTS 174

Chapter 11 DATA REFINEMENT 178
Range of Representations 179
Retrieve Functions 181
Adequacy 183
Refinement of Operations 185
Summary 193
Summary Exercises 194

Chapter 12 MAPPING NOTATION 195
Notation 197
Use in Specifications 201
Data Refinement 205
Recording Equivalence Relations 207
*Arbitrary Sets Modelled on Regular Arrays 212
Summary 214
Summary Exercises 215

Chapter 13 *THEORIES OF DATA TYPES 218
Problem Description 219

Forests 219
Fischer-Galler Algorithm 221
More about Forests 226
Cleaning up Forests 227
Rings 230
Ring Algorithm 231
Summary 233
Summary Exercises 234

Chapter 14 ABSTRACT SYNTAX 235
Notation 238
Use in Specifications 243
*Defining the Abstract Syntax of a Language 247
*Structural Induction 250
Data Refinement 252
Summary 254
Summary Exercises 256

Chapter 15 *ON BEING SUFFICIENTLY ABSTRACT 259
Some Alternative Specifications 259
Implementation Bias 261
Dictionary Example 262
Difficult Refinement Steps 264
Summary 265

Chapter 16 *IMPLICIT DEFINITION OF DATA TYPES 266

Part C THE RIGOROUS METHOD 271

Chapter 17 OVERVIEW OF THE RIGOROUS METHOD 273
A Top-Down View of the Method 274
Language Support for Abstract Data Types 280
On Interfaces 283
Limitations 283

Chapter 18 *EARLEY'S RECOGNIZER 285
Grammars and Parsing 286
Specification 290
First Development Step 292
Second Development Step 295
Third Development Step 296
*Alternative Proof 301
Fourth Development Step 302
Program 305
Modifications 305
Summary 305
Summary Exercises 306

Chapter 19 INPUT/OUTPUT STATEMENTS 307
Definition 307
Proofs 309
Summary 312

Summary Exercises 313

Chapter 20 JOSEPHUS RINGS. 314
Circles 315
Specification 317
Non-Rotation Algorithms 318
Pseudo-Rotation Algorithms. 323

Chapter 21 TELEGRAM ANALYSIS. 325
Henderson's Specification 325
Abstract Specification 326
First Development Step 327
Second Development Step 330
Summary 332
Summary Exercises 332

Chapter 22 ON DESIGN. 333
Programs and Their Data 333
Anticipating Change 337
Input Independence. 338
Environment Independence 340
Summary 340

Appendix A *LANGUAGE DEFINITION. 341

Appendix B DEDUCTION RULES FOR PROOFS. 344
Decomposition Steps 344
Data Refinement. 347

Appendix C PROPERTIES OF OPERATORS. 348
Logic Notation 348
Set Notation 349
List Notation 350
Mapping Notation. 352
Operations 353
Abstract Syntax 353

Appendix D *SELECTED DETAILED PROOFS. 354

GLOSSARY. 357

ANSWERS TO SELECTED EXERCISES. 364

BIBLIOGRAPHY. 371

INDEX. 379

LIST OF ILLUSTRATIONS

Figure 1. Organization 2
Figure 2. Theory versus Practice 9
Figure 3. Cartoon 11
Figure 4. The Rigorous Method 13
Figure 5. Requirements for Specifications 20
Figure 6. Parameter Substitution 22
Figure 7. Implicit Function Specification 24
Figure 8. Meaning of Implicit Function Specification 25
Figure 9. Propositional Operators and Their Priority 27
Figure 10. Bounded Quantified Expression 33
Figure 11. Proof Structure 40
Figure 12. A Binary Tree 49
Figure 13. Evaluation of Factorial 51
Figure 14. Program for Factorial 63
Figure 15. Implicit Operation Definition 66
Figure 16. Multiplication Program 80
Figure 17. Rules for Sequential Statements 81
Figure 18. Picture of Sequential Statement 82
Figure 19. Rules for Conditional Statements 90
Figure 20. Picture of Conditional Statement 90
Figure 21. Rules for Initialized Iteration (Up) 95
Figure 22. Picture of Initialized Iteration (Up) 96
Figure 23. Rules for Initialized Iteration (Down) 101
Figure 24. Picture of Initialized Iteration (Down) 102
Figure 25. Rules for Simple Iteration (Down) 110
Figure 26. Primes Program 117
Figure 27. Spectrum of Abstraction 124
Figure 28. Bill of Materials 125
Figure 29. Data Type 127

Figure 30. Rules for DO/BY/TO Iteration 129
Figure 31. Picture of DO/BY/TO Iteration 130
Figure 32. Set Operators 140
Figure 33. Set Operators 141
Figure 34. Specification of Students in Classroom Problem 143
Figure 35. Properties of Relations over Integers 151
Figure 36. An Equivalence Relation 153
Figure 37. Recording Equivalence Relations 154
Figure 38. List Operators 161
Figure 39. Specification of Stack 165
Figure 40. Multiplication Table Viewed as List 166
Figure 41. Specification of Multiplication Table Problem 166
Figure 42. Extended List Functions 168
Figure 43. Specification of Students who Complete Exercises Problem 175
Figure 44. Data Type Invariant 177
Figure 45. Aspects of Stepwise Development 179
Figure 46. Correspondence of List Representation to Sets 181
Figure 47. Representations Viewed under Retrieve Functions 186
Figure 48. Operation Modelling 186
Figure 49. Rules for Proof of Refinement 187
Figure 50. Collections of Operations 188
Figure 51. Refinement of Students in Classroom Problem 189
Figure 52. Mapping for Students' Whereabouts 195
Figure 53. Mapping Operators 200
Figure 54. Specification of Students' Whereabouts Problem 200
Figure 55. Specification of Equivalence Relation Problem 208
Figure 56. Recording Equivalence Relations 209
Figure 57. Versions of Equivalence Relation Problem 218
Figure 58. A Tree 219
Figure 59. Effect of Operation 222
Figure 60. Equivalence Relation Problem: Map to Key 223
Figure 61. Alternative Trees 227
Figure 62. Rings 230
Figure 63. DO Statement Concrete Syntax 236
Figure 64. A List 239
Figure 65. A Two-Level List 240
Figure 66. A Constructed Object 242
Figure 67. Rational Numbers 244
Figure 68. Specification of Rational Numbers Problem 245
Figure 69. Abstract Form of Factorial Program 248
Figure 70. Overall View of Rigorous Method 275
Figure 71. The Rigorous Method 275
Figure 72. Creating a Specification 278
Figure 73. Refinement 278
Figure 74. Decomposition 279
Figure 75. Class-like Construct 281
Figure 76. Parsing an English Sentence 286
Figure 77. Grammar for Expressions 286

Figure 78. Parse Tree of a Logical Expression 289
Figure 79. Primes Example as Input/Output Operations 308
Figure 80. Rules for Sequential Statements with Output 310
Figure 81. Rules for Iteration with Output 310
Figure 82. Josephus Rings 314
Figure 83. Josephus Rings with Counting Interval of Two 314
Figure 84. A Count Tree 321
Figure 85. Telegram Analysis: Main Program 329
Figure 86. Telegram Analysis: Subroutine 331

Chapter 1

INTRODUCTION

Structure of the Book

The body of the book is divided into three parts. Within each part there are several chapters. Each chapter is broken down into a number of sections and sub-sections. An overview of the organization is given in fig. 1.

The purpose of part A (chapters 2-6) is to bring the reader to the point where he can prove programs correct. Since this idea is probably unfamiliar, it is shown for numerical programs where there are no additional problems of unfamiliar notation. Thus, apart from the notation of logic which is being taught, only the familiar algebra (of numbers) is used.

Part B (chapters 7-16) moves away from numerical algorithmic problems into techniques which can be shown to apply to data manipulating problems. This part of the book discusses how abstract data types can be used in writing specifications and in the development of programs.

These first two parts of the book present the basic techniques; both parts cover specification and then proof; both parts begin formally and then develop a less formal (but rigorous) style of proof when enough practice has been gained to make this safe. Part C (chapters 17-22) fits the various techniques into an overall systematic approach to program design and applies the method to a number of problems.

The book is basically self-contained in that all required notation is taught. Readers who are totally unfamiliar with logic notation would benefit from studying chapters 1-3 and 14-15 of Lipschutz(64). (This is the form of reference used throughout the book: the number in brackets following the first author's name refers to the year of publication. A detailed bibliography is given at the end of the book). Four of the chapters (13, 15, 16, and 18) and a number of sections are marked, in the table of contents, with an asterisk: these present additional material

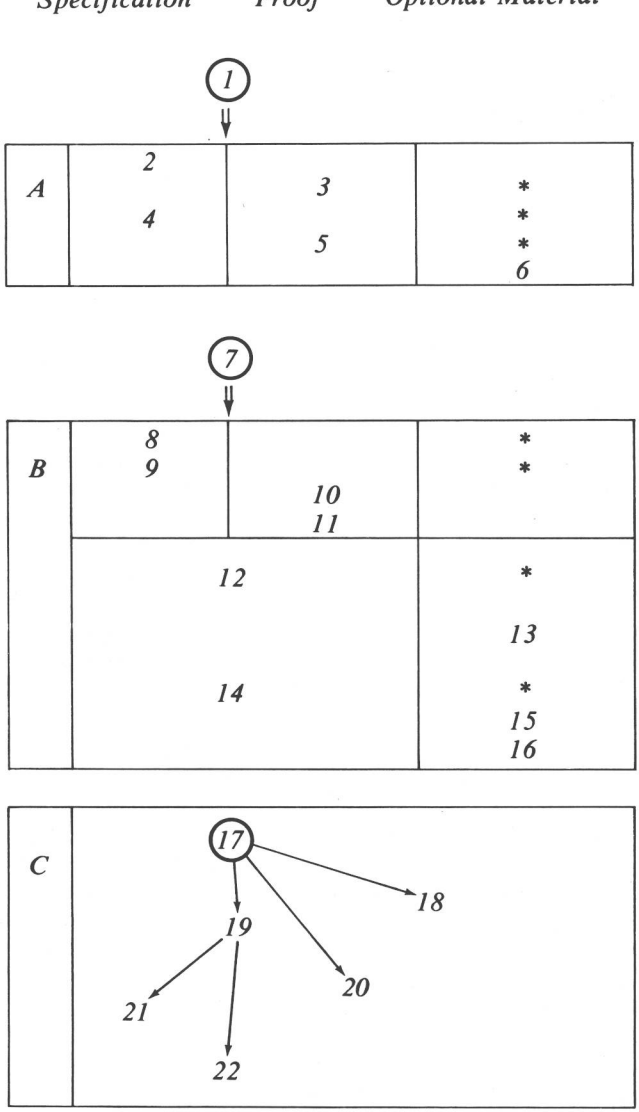

Figure 1 Organization

and are not necessary for a comprehension of the main text. Because it is mainly intended for university use, the optional material is presented in a more condensed form.

The programming language PL/I has been used because it is likely to be familiar to practising programmers. The full language is far from ideal for program proofs. It is, however, in the spirit of the approach of this book to try to use a large language in a constrained way and then to consider the features of the language as required. Here, a subset of PL/I is used which is simple enough for the reader to have no difficulty in translating into, say, Pascal.

At the back of the book there is a glossary of technical terms for reference (terms in this glossary are marked on first reference thus—***predicate***). Similarly, appendices are given which include reference material of use during, and after, reading the book (e.g. appendix C provides access to a number of definitions which are developed as examples or exercises). Many exercises are provided since practice with unfamiliar notation is the only way to gain confidence. Answers are provided for those exercises which check the reader's comprehension of the notation. Most chapters contain a section of summary exercises—these vary in difficulty but are generally more complex than those in the body of the chapters.

Background

The overall requirement for a more effective way of developing software systems is taken for granted here. From the viewpoint of those who wish to see computers assist in solving their problems, there are two major problems: software development is both highly error-prone and is disproportionately costly. The principal bottleneck (both financially and in terms of time) in implementing computer systems is now the production of the software.

The above shortcomings are of concern in all computer applications. As regards freedom from errors, most people would agree that the requirements on, for example, missile early warning, patient monitoring or nuclear power station control systems should be as stringent as possible. Leaving aside such emotive examples one moves into an area where the penalty for failure is easier to translate into financial terms. Much of modern business life relies on computers. If the payroll program fails, it is no longer practical to think in terms of manual backup. In some industries, the cost of the failure could be a strike and attendant loss of production. When assessing the cost of a program failure, such items must be considered as well as expenses related to lost files or security breaches.

Programmers do not usually work in a vacuum. Whoever is paying the salaries of programmers today has cause for complaint at the cost of creating programs.

Having identified that a problem exists, the next step towards its solution is the location of some plausible cause. In the case of software development, the cause is not hard to find. The scale of the applications