国外优质职业教育资源教学用书

程序设计基础

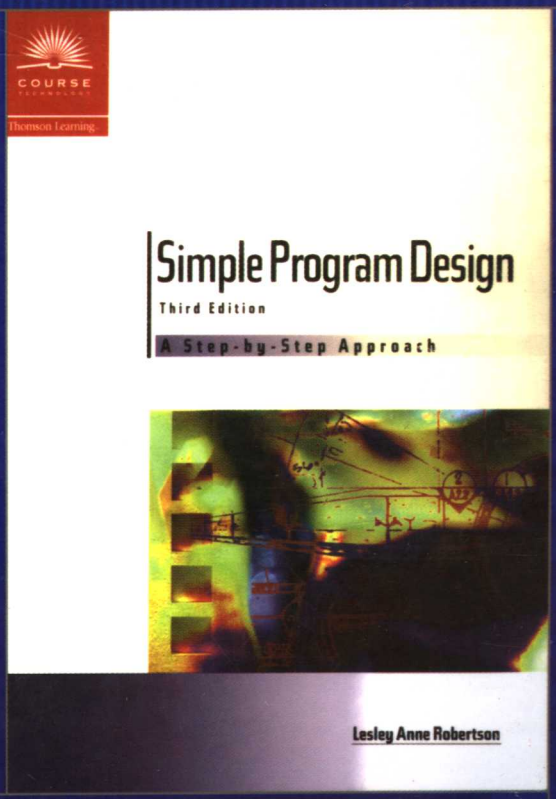
第3版 双语版权：主教材

Simple Program Design

A Step-by-Step Approach

Third Edition

Lesley Anne Robertson



高等教育出版社
Higher Education Press

国外优质职业教育资源教学用书

程序设计基础

（第3版　双语版权：主教材）

Simple Program Design

A Step-by-Step Approach

Third Edition

Lesley Anne Robertson

高等教育出版社

图字：01-2003-3468 号

Simple Program Design A Step-by-Step Approach
Third Edition
Lesley Anne Robertson

图书在版编目(CIP)数据

程序设计基础＝Simple Program Design—A Step-By-Step Approach, Third Edition：第3版 /（澳）罗伯逊（Robertson, L. A.）著. —北京：高等教育出版社，2003.9
ISBN 7-04-012743-1

Ⅰ.程… Ⅱ.罗… Ⅲ.程序设计—英文 Ⅳ.TP311.1

中国版本图书馆 CIP 数据核字(2003)第 061753 号

出版发行	高等教育出版社	**购书热线**	010－64054588
社　　址	北京市西城区德外大街 4 号	**免费咨询**	800－810－0598
邮政编码	100011	**网　　址**	http://www.hep.edu.cn
总　　机	010－82028899		http://www.hep.com.cn
经　　销	新华书店北京发行所		
印　　刷	北京外文印刷厂		
开　　本	787×1092　1/16	**版　　次**	2003 年 9 月第 1 版
印　　张	23.75	**印　　次**	2003 年 9 月第 1 次印刷
字　　数	540 000	**定　　价**	30.70 元

出版说明

2002年7月，国务院召开了全国职业教育工作会议。为了贯彻职教会关于“积极引进国外优质的职业教育资源”的精神，落实教育部周济部长和王湛副部长关于引进国外职业教育教材是“坚持开放要有新局面的一个好举措”、“要把这项工作放在全国职教教材的大体系中全面考虑和规划”的重要批示，高等教育出版社在教育部高等教育司、职业教育与成人教育司的大力支持下，准备系列出版“国外优质职业教育资源教学用书”，包括专业教材、实训教材以及具有鲜明职业教育特色的课程模式教材。信息类引进教材作为引进教材系列的第一批，共有6本书。这套教材的特点是：

（1）专家评选，具有权威性。高等教育出版社邀请清华大学等知名院校信息领域专家以及北京联合大学信息学院等国家级“十五” 规划IT领域重点课题承担学校、教育部高职高专精品专业建设项目院校的专家、教授，成立了引进教材专家组，设立“国外优质职业教育教材引进与借鉴”研究课题，从国外教材与国内职业教育教材的比较和专业适用性等方面进行研究，经过认真遴选和评议，从汤姆森学习出版集团（Thomson Learning）推荐的60余本教材中精选出了这套教材，具有一定的权威性。

（2）采用最新版本，注重体现高职高专教育特色和教材的立体化建设思想。比如，《New Perspectives on Computer Concepts 6ed》，中译名为《计算机导论（第6版）》，这本书是2003年2月最新出版的，最新版本的采用有利于学生了解IT业的最新动向和最新技术；精选的每本教材都配备了大量的应用实例、上机实验、互动操作练习及测试等内容，体现了国外职业教育教材的显著特色；除引进主教材外，还相应地引进了教学辅助光盘、CAI课件、网上教学资源等其他教学资源，力图为使用者提供一套完整的教学解决方案。

（3）采用双语版权引进方式。在对全书进行原版影印的同时，配有针对主要内容的中文编译。这样，既保持了原版教材的“原汁原味”，又便于教师和学生阅读和理解。此外，在引进教材过程中遵循质优和价廉的原则，虽然有些教材采用彩色印刷，但在价格上与国内职业教育教材相当，不给教师和学生增加更多的经济负担。

引进国外优质的职业教育资源是高等教育出版社职业教育教材建设新的尝试，是迈向国际化的重要一步。今后，我们在引进国外教材版权的基础上，还要进一步引进国外优秀的作者资源，加快我国职业教育与国际接轨的步伐。同时，我们还将继续努力，将更多的引进教材推荐给高职高专院校，为促进中国高职高专教育的发展贡献力量。

高等教育出版社

2003年5月

Preface

With the increased popularity of programming courses in our universities, colleges and technical institutions, there is a need for an easy-to-read textbook on computer program design. There are already dozens of introductory programming texts using specific languages such as PASCAL, BASIC, COBOL or C, but they usually gloss over the important step of designing a solution to a given programming problem.

This textbook tackles the subject of program design by using structured programming techniques and pseudocode to develop a solution algorithm. The recommended pseudocode has been chosen because of its closeness to written English, its versatility and ease of manipulation, and its similarity to the syntax of most structured programming languages.

Simple Program Design is designed for programmers who want to develop good programming skills for solving common business problems. Too often, programmers who are faced with a problem launch straight into the code of their chosen programming language, instead of concentrating on the actual problem at hand. They become bogged down with the syntax and format of the language, and often spend many hours getting the program to work. Using this textbook, the programmer will learn how to define the problem, how to design a solution algorithm, and how to prove the algorithm's correctness, before coding a single statement from any programming language. By using pseudocode and structured programming techniques, the programmer can concentrate on developing a well-designed and correct solution, and thus eliminate many frustrating hours at the testing phase.

The book is divided into thirteen chapters, beginning with a basic explanation of structured programming techniques, top-down development and modular design. Then, concept by concept, the student is introduced to the syntax of pseudocode; methods of defining the problem; the application of basic control structures in the development of the solution algorithm; desk-checking techniques; arrays; hierarchy charts; module design; parameter passing; object-oriented design methodology; and many common algorithms.

Each chapter thoroughly covers the topic at hand, giving practical examples relating to business applications, and a consistently structured approach when representing algorithms and hierarchy charts.

This third edition of *Simple Program Design* contains additional material which complements the information provided in the first two editions. A new chapter on array processing has been included, covering single and multi-dimensional arrays, together with common operations on arrays and algorithms for their manipulation. Modularisation has been extended to cover two chapters, with communication between modules and parameter passing introduced at an earlier stage in the design process.

Two new chapters covering object-oriented design methodology have been included in this third edition. These chapters were written by Kim Styles and Wendy Doube, lecturers in computing at the Gippsland School of Computing and Information Technology, Monash University. They introduce the concepts of object-oriented design and the steps involved in creating an object-oriented solution to a problem. Step-by-step algorithms using OO design are provided, as well as material on multiple objects and interfaces.

Many courses now require students to be proficient in more than one algorithm design technique. As in the first two editions, pseudocode has been chosen as the main algorithm design technique throughout the book. However, this third edition now offers two alternate methods of representing algorithms: flowcharts, in Appendix 1, and Nassi-Schneiderman diagrams, in Appendix 2. All algorithms developed in pseudocode in Chapters 2, 3, 4 and 5 have been presented again in Appendix 1 — using flowcharts — and Appendix 2 — using Nassi-Schneiderman diagrams.

This third edition also provides ten programming problems, of increasing complexity, at the end of each chapter, so that teachers now have a choice of exercises that matches the widely varying abilities of their students.

I would like to thank Kim Styles and Wendy Doube, lecturers in Computing at Monash University, for their wonderful input on object-oriented design methodology, Paul Moriarty, of Saranac Lake, New York, for his enthusiastic suggestions, and my brother, Rick Noble, for his amusing cartoons at the beginning of each chapter.

Lesley Anne Robertson

The Author

Lesley Anne Robertson was introduced to structured programming techniques and top-down design when she joined IBM, Australia, in 1973 as a trainee programmer. Since then, she has consistently used these techniques as a programmer, a systems analyst, and finally a Lecturer in Computing at the University of Western Sydney, NSW, where she taught computer programming for eleven years.

Lesley now lives on a vineyard and winery in Mudgee, NSW, with her husband, David, and daughters, Lucy and Sally.

Contents

4 Selection control structures

Expands the selection control structure by introducing multiple selection, nested selection, and the case construct in pseudocode. Several algorithms, using variations of the selection control structure, are developed.

5 Repetition control structures

Develops algorithms which use the repetition control structure in the form of DOWHILE, REPEAT...UNTIL, and counted repetition loops.

6 Pseudocode algorithms using sequence, selection and repetition

Develops algorithms to eight simple programming problems using combinations of sequence, selection and repetition constructs. Each problem is properly defined; the control structures required are established; a pseudocode algorithm is developed; and the solution is manually checked for logic errors.

7 Array processing

Introduces arrays, operations on arrays, and algorithms which manipulate arrays. Algorithms for single and two-dimensional arrays, which initialise the elements of an array, search an array and write out the contents of an array, are presented.

8 First steps in modularisation

Introduces modularisation as a means of dividing a problem into subtasks. Hierarchy charts and parameter passing are introduced and several algorithms which use a modular structure are developed.

9 Further modularisation, cohesion and coupling

Develops modularisation further, using a more complex problem. Module cohesion and coupling are introduced, several levels of cohesion and coupling are described and pseudocode examples of each level are provided.

10 General algorithms for common business problems

Develops a general pseudocode algorithm for four common business applications. All problems are defined; a hierarchy chart is established; and a pseudocode algorithm is developed, using a mainline and several subordinate modules. The topics covered include report generation with page break, a single-level control break, a multiple-level control break, and a sequential file update program.

11 Object-oriented design

Introduces object-oriented design, objects, classes, attributes, methods and information hiding. The steps required to create an object-oriented solution to a problem are provided and solution algorithms developed.

12 More object-oriented design

Introduces the concept of multiple classes, polymorphism and method overriding in object-oriented design. Discusses the relationship between classes and lists the steps required to create an object-oriented design to a problem with multiple classes.

13 Conclusion

A revision of the steps involved in good top-down program design.

Appendix 1 Flowcharts

Introduces flowcharts for those students who prefer a more graphic approach to program design. Algorithms which use a combination of sequence, selection and repetition are developed in some detail.

Appendix 2
Nassi-Schneiderman diagrams

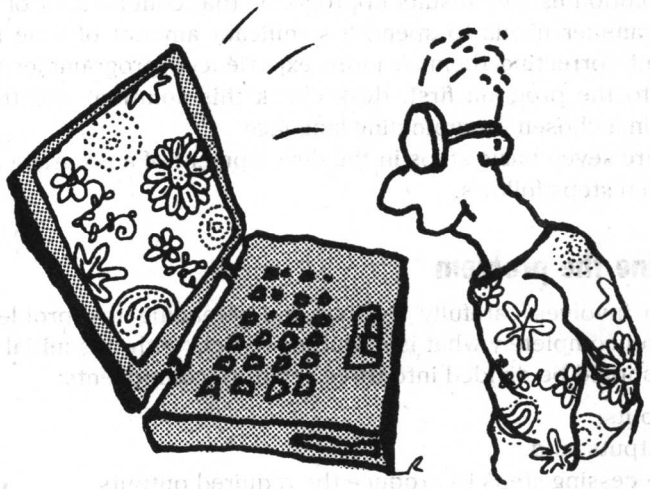
Introduces Nassi-Schneiderman diagrams for those students who prefer a more diagrammatic approach to program design. Algorithms which use a combination of sequence, selection and repetition constructs are developed in some detail.

Appendix 3
Special algorithms

Contains a number of algorithms which are not included in the body of the textbook and yet may be required at some time in a programmer's career.

CHAPTER 1

Program design



Objectives

- To describe the steps in the program development process
- To explain structured programming
- To introduce algorithms and pseudocode
- To describe program data

Outline

1.1 STEPS IN PROGRAM DEVELOPMENT

Computer programming is an art. Many people believe that a programmer must be good at mathematics, have a memory for figures and technical information, and be prepared to spend many hours sitting at a computer, typing programs. However, given the right tools, and steps to follow, anyone can write well-designed programs. It is a task worth doing, as it is both stimulating and fulfilling.

Programming can be defined as the development of a solution to an identified problem, and the setting up of a related series of instructions which, when directed through computer hardware, will produce the desired results. It is the first part of this definition that satisfies the programmer's creative needs: that is, to design a solution to an identified problem. Yet this step is so often overlooked. Leaping straight into the coding phase without first designing a proper solution usually results in programs that contain a lot of errors. Often the programmer needs to spend a significant amount of time finding these errors and correcting them. A more experienced programmer will design a solution to the program first, desk check this solution, and then code the program in a chosen programming language.

There are seven basic steps in the development of a program. An outline of these seven steps follows.

1 Define the problem

This step involves carefully reading and rereading the problem until you understand completely what is required. To help with this initial analysis, the problem should be divided into three separate components:

- The inputs,
- The outputs, and
- The processing steps to produce the required outputs.

A defining diagram as described in Chapter 3 is recommended in this analysis phase, as it helps to separate the define the three components.

2 Outline the solution

Once the problem has been defined, you may decide to break the problem up into smaller tasks or steps, and establish an outline solution. This initial outline is usually a rough draft of the solution which may include:

- The major processing steps involved,
- The major subtasks (if any),
- The major control structures (e.g. repetition loops),
- The major variables and record structures, and
- The mainline logic.

The solution outline may also include a hierarchy or structure chart. The steps involved in creating this outline solution are detailed in Chapters 2 to 6.

3 Develop the outline into an algorithm

The solution outline developed in Step 2 is then expanded into an algorithm: a

set of precise steps that describe exactly the tasks to be performed and the order in which they are to be carried out. This book uses pseudocode (a form of structured English) to represent the solution algorithm, as well as structured programming techniques. Flowcharts and Nassi-Schneiderman diagrams are also provided in Appendix 1 and Appendix 2 for those who prefer a more pictorial method of algorithm representation. Algorithms using pseudocode and the Structure Theorem are developed thoroughly in Chapters 2 to 7.

4 Test the algorithm for correctness

This step is one of the most important in the development of a program, and yet it is the step most often forgotten. The main purpose of desk checking the algorithm is to identify major logic errors early, so that they may be easily corrected. Test data needs to be walked through each step in the algorithm to check that the instructions described in the algorithm will actually do what they are supposed to. The programmer walks through the logic of the algorithm, exactly as a computer would, keeping track of all major variables on a sheet of paper. Chapter 3 recommends the use of a desk check table to desk check the algorithm, and many examples of its use are provided.

5 Code the algorithm into a specific programming language

Only after all design considerations have been met in the previous four steps should you actually start to code the program into your chosen programming language.

6 Run the program on the computer

This step uses a program compiler and programmer-designed test data to machine test the code for syntax errors (those detected at compile time) and logic errors (those detected at run time). This is usually the most rewarding step in the program development process. If the program has been well designed, the usual time-wasting frustration and despair often associated with program testing are reduced to a minimum. This step may need to be performed several times until you are satisfied that the program is running as required.

7 Document and maintain the program

Program documentation should not be listed as the last step in the program development process, as it is really an ongoing task from the initial definition of the problem to the final test result.

Documentation involves both external documentation (such as hierarchy charts, the solution algorithm, and test data results) and internal documentation that may have been coded in the program. Program maintenance refers to changes which may need to be made to a program throughout its life. Often these changes are performed by a different programmer from the one who initially wrote the program. If the program has been well designed using structured programming techniques, the code will be seen as self documenting, resulting in easier maintenance.

1.2 STRUCTURED PROGRAMMING

Structured programming helps you to write effective, error-free programs. The original concept of structured programming was set out in a paper published in 1964 in Italy by Bohm and Jacopini. They established the idea of designing programs using a Structure Theorem based on three control structures. Since then a number of authors, such as Edsger Dijkstra, Niklaus Wirth, Ed Yourdon and Michael Jackson, have developed the concept further and have contributed to the establishment of the popular term 'structured programming'. This term now refers not only to the Structure Theorem itself, but also to top-down development and modular design.

Top-down development

Traditionally, programmers presented with a programming problem would start coding at the beginning of the problem and work systematically through each step until reaching the end. Often they would get bogged down in the intricacies of a particular part of the problem, rather than considering the solution as a whole. In the top-down development of a program design, a general solution to the problem is outlined first. This is then broken down gradually into more detailed steps until finally the most detailed levels have been completed. It is only after this process of 'functional decomposition' (or 'stepwise refinement') that the programmer starts to code. The result of this systematic, disciplined approach to program design is a higher precision of programming than was possible before.

Modular design

Structured programming also incorporates the concept of modular design, which involves grouping tasks together because they all perform the same function (e.g. calculating sales tax or printing report headings). Modular design is connected directly to top-down development, as the steps or subtasks into which the programmer breaks up the program solution will actually form the future modules of the program. Good modular design aids in the reading and understanding of the program.

The Structure Theorem

The Structure Theorem revolutionised program design by eliminating the GOTO statement and establishing a structured framework for representing the solution. The theorem states that it is possible to write any computer program by using only three basic control structures. These control structures are:

- Sequence;
- Selection, or IF-THEN-ELSE; and
- Repetition, or DOWHILE.

They are covered in detail in Chapter 2.

1.3 AN INTRODUCTION TO ALGORITHMS AND PSEUDOCODE

Structured programming techniques require a program to be properly designed before coding begins, and it is this design process that results in the construction of an algorithm.

What is an algorithm?

An algorithm is like a recipe: it lists the steps involved in accomplishing a task. It can be defined in programming terms as a set of detailed, unambiguous and ordered instructions developed to describe the processes necessary to produce the desired output from a given input. The algorithm is written in simple English and is not a formal document. However, to be useful, there are some principles which should be adhered to. An algorithm must:

- be lucid, precise and unambiguous;
- give the correct solution in all cases; and
- eventually end.

For example, if you want to instruct someone to add up a list of prices on a pocket calculator, you might write an algorithm such as the following:

```
Turn on calculator
Clear calculator
Repeat the following instructions
    Key in dollar amount
    Key in decimal point (.)
    Key in cents amount
    Press addition (+) key
Until all prices have been entered
Write down total price
Turn off calculator
```

Notice that in this algorithm the first two steps are performed once, before the repetitive process of entering the prices. After all the prices have been entered and summed, the total prices can be written down and the calculator can be turned off. These final two activities are also performed only once. This algorithm satisfies the desired list of properties: it lists all the steps in the correct order from top to bottom, in a definite and unambiguous fashion, until a correct solution is reached. Notice that the steps to be repeated (entering and summing the prices) are indented, both to separate them from those steps performed only once and to emphasise the repetitive nature of their action. It is important to use indentation when writing solution algorithms because it helps to differentiate between the three control structures.

What is pseudocode?

Pseudocode, flowcharts and Nassi-Schneiderman diagrams are all popular ways of representing algorithms. Flowcharts and Nassi-Schneiderman diagrams are covered in Appendix 1 and Appendix 2 of this text, while pseudocode has been chosen as the primary method of representing an algorithm because it is easy to read and write. Pseudocode is really structured English. It is English that has been formalised and abbreviated to look very like high-level computer languages.

There is no standard pseudocode at present. Authors seem to adopt their own special techniques and sets of rules, which often resemble a particular programming language. This book attempts to establish a standard pseudocode for use by all programmers, regardless of the programming language they choose. Like many versions of pseudocode, this version has certain conventions, as follows:

1 Statements are written in simple English.
2 Each instruction is written on a separate line.
3 Keywords and indentation are used to signify particular control structures.
4 Each set of instructions is written from top to bottom, with only one entry and one exit.
5 Groups of statements may be formed into modules, and that group given a name.

Pseudocode has been chosen to represent the solution algorithms in this book because its use allows the programmer to concentrate on the logic of the problem.

1.4 PROGRAM DATA

Because programs are written to process data, you must also have a good understanding of the nature and structure of the data being processed. Data within a program may be a single variable, such as an integer or a character; or a group item (sometimes called an aggregate), such as an array, or a file.

Variables, constants and literals

A variable is the name given to a collection of memory cells, designed to store a particular data item. It is called a variable because the value stored in that variable may change or vary as the program executes. For example, the variable total_amount may contain several values during the execution of the program.

A constant is a data item with a name and a value that remain the same during the execution of the program. For example, the name fifty may be given to a data item that contains the value 50.

A literal is a constant whose name is the written representation of its value. For example, the program may contain the literal '50'.

Elementary data items

An elementary data item is one containing a single variable that is always treated as a unit. These data items are usually classified into data types. A data