Michael Jackson

System Development

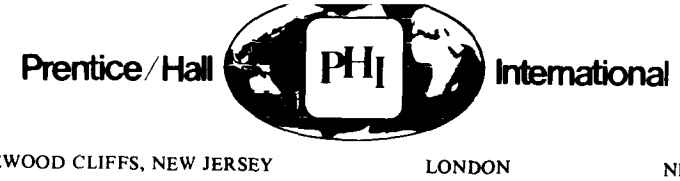
PRENTICE-HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE

C.A.R. HOARE SERIES EDITOR

SYSTEM DEVELOPMENT

M. A. JACKSON
Michael Jackson Systems Limited

based on the work of
M. A. JACKSON *and* J. R. CAMERON



ENGLEWOOD CLIFFS, NEW JERSEY LONDON NEW DELHI
SINGAPORE SYDNEY TOKYO TORONTO WELLINGTON

To Judy, Daniel, Tim, David and Adam

Library of Congress Cataloging in Publication Data
Jackson, Michael, 1936–
System development.

Bibliography: p.
Includes index.
1. System design. I. Title.
QA76.9.S88J33 003 82-619
ISBN 0-13-880328-5 AACR2

British Library Cataloguing in Publication Data
Jackson, Michael
System development.
1. Systems engineering 2. Business
I. Title
658.4'032 TA168

ISBN 0-13-880328-5

ISBN 0-13-880328-5

PRENTICE-HALL INTERNATIONAL INC., London
PRENTICE-HALL OF AUSTRALIA PTY., LTD., Sydney
PRENTICE-HALL CANADA, INC., Toronto
PRENTICE-HALL OF INDIA PRIVATE LIMITED, New Delhi
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

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Preface

1 SCOPE OF JSD

This book is about a system development method. The method, known by the acronym JSD, is a method for specifying and implementing computer systems. We use the general term 'development' to cover a range of activities usually carried out by people whose job titles are 'systems analyst', 'system designer', 'program designer' or 'programmer'. These activities include requirements specification, functional specification, logical system design, application system design, physical system design, program specification and design, program implementation, and system and program maintenance. The book is addressed to people, whatever their job titles, who engage in any of these activities, or want to understand how they may be approached.

There is very little agreement on the definitions of these activities, and even less on the meanings of the job titles 'analyst', 'designer' and 'programmer'. JSD cuts across even the very small area of agreement that does exist, redrawing the boundaries between activities and between jobs, so that the existing names and titles become almost entirely inappropriate. So we shall refer to the activities by the general term 'development', and we shall use the title 'developer' to mean whoever is doing the activity currently under discussion.

Within JSD the primary distinction is between specification and implementation. The JSD development procedure has six steps, of which the first four are concerned with creating a specification of the required system, and the last two with implementing that specification. What is often called 'design' has largely been absorbed into the implementation part of JSD. This division of development into two major parts—specification and implementation—seems to be beneficial in many ways. One of the most important of these is that it encourages recognition of what has arguably always been the major division between people working in system development: the division between those whose primary interest lies with the system user and those whose primary interest lies with the computer.

JSD does not encompass every activity associated with system development. It excludes activities such as project selection, project planning and management, and cost/benefit analysis; it excludes procedures for system acceptance, installation and cutover; it excludes the whole area of human engineering in such matters as dialog

design. JSD also excludes specialized application skills. In a system concerned with controlling inventory, JSD has nothing to say about the question of whether it would be better to provide an improved service to customers or to reduce the size of the total inventory. For example, in a system concerned with controlling elevators in a building, JSD provides no guidance towards a decision whether the pattern of elevator movements should follow one algorithm or another, or whether the system should provide gentle music to soothe the impatience of waiting customers.

2 THE SYSTEM AND THE REAL WORLD

The word 'system' can be used extensively, to include computer procedures, manual procedures, all or part of the user organization, everything and everyone that directly affects or is affected by the result of the development work. So, in an application concerned with control of aircraft engines the system includes the engines themselves and the devices which increase or reduce flow of air or fuel; in a purchasing application, the system includes the suppliers, the delivery trucks, the suppliers' documents, the receiving bays, the warehouse locations, the clerks in the purchasing department.

In JSD we restrict the use of the word 'system', referring essentially to what is created by the development activity; we distinguish the system from the real world outside. The real world provides the subject matter for the system: it contains the engines to be controlled, the employees to be paid, the customers and suppliers whose transactions are to be accounted for. The system itself consists of computer and manual procedures and hardware; we think of it as having a definite boundary—the system boundary—across which inputs and outputs flow between the real world and the system. This view is pictured in Fig. 1. In an old-fashioned batch data processing system, the system boundary was almost a physical boundary, enclosing the data preparation and the computer departments, and the mail room, together with some clerical departments that interfaced with data preparation on one side and the organization's customers, employees and suppliers on the other. In on-line and embedded systems, the boundary is less tangible, but still there.

In JSD the real world is regarded as given, a fixed starting point. This view reflects the exclusion from JSD of specific application knowledge; it is no part of JSD to choose the most economic policy for stock replenishment, or to negotiate with labor unions to determine rules for overtime payments, or to decide the algorithm for computing the movement of an airplane's control surfaces to correct a deviation from course. Our concern in JSD is to ensure that the system correctly reflects the real world as it is, and to provide the functions requested by the user, to a specification in which the user has the determining voice.

Although we regard the real world as given, we do not, of course, exclude the possibility that some or all of the real world must be invented or changed. For the most part, the invention or change is not itself an integral part of the JSD development. An extreme illustration is the general functional requirement: develop a good video game. The appropriate expert—if one exists—might determine that a certain psychological pattern would be good, a certain progression of success and reward, expectation and fulfilment, tension and satisfaction; and that the game should be about an intricate and difficult search for some treasure. Here, then, is the real world for this system. The JSD developer helps to specify this real world, working in cooperation with the expert and

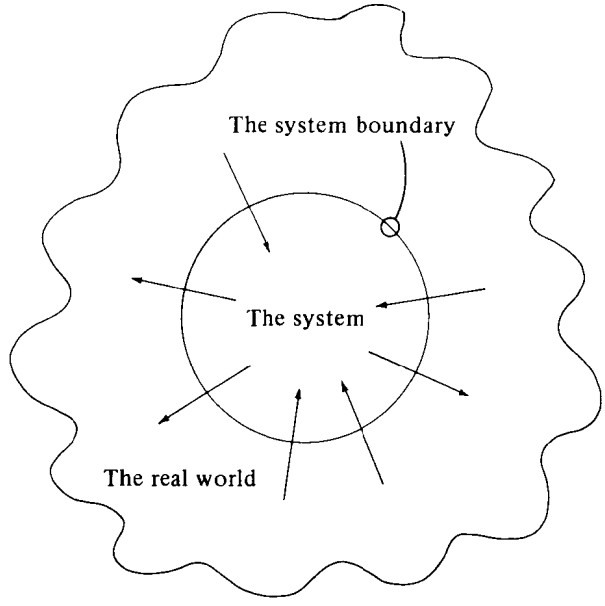


Fig. 1

providing concepts and notations in which the specification can be clearly expressed. The expert does the invention, and the JSD developer does the description. At a later stage of development, especially in the implementation stage, some technical invention is necessary within the system; that is a part of the developer's task.

3 APPLICABILITY OF JSD

JSD is used to develop systems whose subject matter has a strong time dimension. For example, the subject matter of the elevator control system is the real elevators, the real customers, the buttons they press to summon the elevator, and so on. To describe this subject matter properly we must place a major emphasis on its time dimension: the customer presses the call button before the elevator arrives; the doors open after the elevator arrives; the elevator leaves the floor after the doors close. In a data processing system for a bank, the subject matter is the real customers, their checks, their bank loans, their repayments, the interest charges, and so on. Here again, the time dimension is of central importance. A loan must be granted before repayments begin; repayments are credited before interest charges are applied; the checking account is opened before any checks are written.

Because of its emphasis on the time dimension, JSD is very widely applicable. Certainly it may be used for developing embedded systems, switching systems, control systems, and all kinds of data processing systems, both on-line and batch. All of these are concerned with real worlds in which the time dimension is of central importance.

4 JSD AND JSP PRINCIPLES

The system development method JSD has grown out of JSP, a program design method which has been widely taught and used over the past ten years. JSD may be seen as an enlargement of JSP, applying the same principles to a larger class of problems and to a larger part of the development task.

The starting point for JSP is a full specification of the program to be designed. A JSP program is a sequential process, to be eventually implemented in a sequential programming language such as PL/I, COBOL, Fortran, Pascal, or assembler language. Its inputs and outputs are viewed as sequential streams of records: a magnetic tape file or a line printer report is obviously a sequential stream of records; but so also is a set of segments retrieved from a database, or a set of terminal input messages in a conversation. A simplified form of the JSP program design procedure is:

(a) describe the structure of each input and output data stream;
(b) combine these structures to form a program structure;
(c) list the operations which the program must execute to produce the outputs from the inputs, and allocate each operation to its proper place in the program structure;
(d) write the program text, adding the necessary conditions to control execution of iteration components (loops or repetitions) and selection components (if–else or case constructs).

An essential principle of JSP is that the subject matter of the computation is described first, in steps (a) and (b), while the detailed function of the program is dealt with later, in step (c). This principle is central also in JSD. The subject matter in JSD is the real world, strongly ordered in time, outside the system; the detailed function is the production of system outputs. In JSD the early development steps are concerned with describing the real world, and explicit consideration of system function is deferred until later.

A program to satisfy a realistic specification usually needs to be decomposed into two or more sequential processes: at step (b) of the JSP design procedure, the developer recognizes that there are structural conflicts, or clashes, among the descriptions of the data streams. JSP provides a classification of these structure clashes into boundary, ordering, and interleaving clashes. For each type of clash there is a prescribed decomposition into a pattern of sequential processes connected by data streams internal to the program.

JSD does not start from a given specification, nor does it decompose the system into sequential processes. Instead, JSD development begins by creating a specification for the system, building it up from parts which are themselves sequential processes: the activity is therefore one of synthesis rather than decomposition.

Where a program is decomposed into connected sequential processes, JSP provides an implementation technique for recombining these processes into a single executable program. This technique, known as program inversion, is a transformation of a sequential process text into the text of an equivalent subroutine; the processes can then be combined into a hierarchical structure of subroutines. Combining processes in this way is regarded as an implementation activity, to be carried out at the end of program development: the hierarchical structure is devised purely for purposes of efficient execution, and is not considered in the earlier design steps.

In the same way, JSD provides an implementation technique, based on program inversion and other transformations, for combining the sequential processes of the system specification into an efficiently executable system.

5 SYSTEM DEVELOPMENT AND PROGRAM DESIGN

Conventionally, program design and programming are thought of as the final stages in system development. The earlier activities of system analysis, specification, and design produce a higher-level structure of the system whose lower levels remain to be completed by programmers. From this point of view, organizations that have adopted JSP as their program design method might hope to find that JSD is a 'front-end' to JSP: JSD would create the specifications for programs which would then be designed using JSP.

But it is not so. Aspects of JSP are diffused through the JSD development procedure. The early steps of JSP are concerned with the structure of sequential processes. They are directly relevant to the development of a JSD specification, which is composed of sequential processes. The implementation techniques of JSP are embodied in those of JSD. The JSP identification of structure clashes has a part to play in the JSD description of its real world subject matter. Certainly, there are some parts of a JSD specification—especially batch reporting programs and batch or on-line treatment of the system inputs—which can be handed over to a JSP program designer in the traditional way; and there are tasks in the JSD implementation stage which are very close to conventional program design tasks. But much of the traditional work of program design has already been incorporated in the development of a JSD specification. This is one example of the way in which JSD redefines the boundaries between development activities.

It is not necessary to know JSP to read this book. Those parts of JSP which are incorporated into JSD are explained as they arise, and JSP notations are included, with JSD notations, in an appendix. A full account of JSP from a program design point of view is given in M. A. Jackson, *Principles of Program Design*, Academic Press; another account is given in Leif Ingevaldsson, *JSP: A Practical Method of Program Design*, Studentlitteratur.

6 ARRANGEMENT OF THIS BOOK

The book is arranged in three major parts. In Part I, the JSD development method is introduced. The underlying principles of JSD are explained and justified by general arguments in an informal way; the development procedure, divided into six steps, is described, and is illustrated by a tiny example. Readers who want only a broad understanding of JSD may find that this part of the book is sufficient for their needs.

Part II is a series of chapters following the sequence of the JSD development steps. Three applications, introduced at the beginning of Part II, are used to provide illustration of the points arising in each development step. This interleaved treatment of the three applications allows technical material to be placed where it arises most naturally. To help the reader to follow the development of each application, the main results for each are gathered together in appendices and should be referred to as necessary to re-establish the context of the application as it progresses.

Part III contains material on some general aspects of development, such as errors and system maintenance, and a retrospective view of JSD in the light of the material in this part and in Part II.

In addition to the appendices, there is a glossary of JSD terms and a summary of notations used in JSD and in JSP at the end of the book.

It is hoped that the repeated discussion of main points of principle and practice will help those readers to whom the ideas are unfamiliar and hard to grasp, without irritating those for whom they are more quickly and easily digestible.

7 EXAMPLES

The examples used are very small; certainly they are smaller than any system that is likely to be put into productive use. Even in these small examples, some detail, especially of documentation, has been omitted from the discussion.

Of course, this is inevitable. In a book of this size, there is too little room for even one complete realistic example. Some authors solve this difficulty by presenting examples of a realistic size but limiting their discussion to fragments rather than complete examples; this is often done where the method presented is top-down. But JSD is not a top-down method, and in a section at the end of the book we discuss why we believe that top-down approaches are essentially inadequate. We prefer instead to limit ourselves to small examples which can be treated fully, omitting nothing that the reader might find difficult to supply or that might invalidate the treatment given.

We hope that the reader will not conclude that JSD is a method for developing very small, toy, systems. E. W. Dijkstra once observed that it is hard to give a talk: if you use no examples, most of the listeners will fail to understand the subject; if you do use examples, many of the listeners will conclude that the examples are themselves the subject. The subject of JSD is the development of a wide class of systems, including very large systems. The examples given are small, because they are intended to illustrate the exposition of the method, not to delimit its applicability.

Acknowledgements

The ideas in this book take their form from work done by John Cameron and myself over the past four years. We have worked together both in developing the ideas and in presenting them on courses, so in that sense the book is as much his as mine. He has also commented extensively and carefully on an earlier draft of the book, saving me from many mistakes: but the words are mine, so any defects of detail and expression are mine too.

I have never found it easy to recognize, and so to acknowledge, the sources of ideas I have worked on and promulgated. The influence of Tony Hoare's work on communicating sequential processes will be obvious, and also the influence of Rod Burstall and John Darlington's work on program transformation. But I have also been influenced, without doubt, by many other people. A paper presented at a conference, a book partly read, an informal discussion at a meeting, can all impart the germ of an idea without a conscious awareness of the debt. I hope that anyone who recognizes the influence of his own work in this book will accept these words as an explanation and an apology.

Cliff Jones and Mike McKeag have given me generous encouragement and valuable comments on an earlier draft of the book. So too have Barry Dwyer and Hans Nägeli. Tony Hoare, the editor of this series, first invited me to write the book, and then sustained me with encouragement and support when I needed it most. Among the few people who have made fundamental advances in software engineering, he is distinguished by his readiness to listen seriously, with care and interest, to ideas that are so much less exact and elegant than his own. Without his help this book would not have been written.

Contents

PART I

INTRODUCTION TO JSD

This part of the book is arranged in four chapters. The first three chapters explain and discuss the fundamental principles of JSD, and the fourth chapter describes the JSD development procedure.

Chapter 1 is about the distinction between model and function and the priority given to modelling over functional considerations in JSD. In JSD, development does not begin by specifying the function of the system to be developed: instead, it begins by describing and modelling the real world which provides the subject matter of the system. The functions of the system are specified on the basis of this model. A JSD system may be regarded as a simulation of the relevant parts of the real world outside the system; system functions are regarded as providing outputs derived from the behavior of this simulation.

Chapter 2 is about process models. The JSD method is concerned with applications in which the real world is dynamic, with events occurring in time-ordered sequence. Most data processing applications are of this kind, as also are applications in process control, embedded systems, and switching systems. To model such a dynamic reality adequately, it is necessary to use a modelling medium which is itself dynamic. JSD models are therefore built from sequential processes, rather than from the components of a database. The term 'sequential process' has a strong flavor of programming and technology: but in essence a sequential process is no more than a statement of the time-ordering of events.

Chapter 3 is about implementation. Because a JSD specification, both of model and of function, is expressed in terms of sequential processes, it is in principle capable of direct execution on the computer (or computers) on which the system is to run. But there is often—in typical data processing systems, always—a mismatch between the number and characteristics of the processes in the JSD specification and the number and characteristics of those which can be conveniently executed on readily available computers and operating systems. The implementation step in JSD is therefore centrally concerned with transforming the specification to make it convenient to execute. By regarding the implementation task as one of transformation, we narrow the gap between the specification and the executable system which implements it.

Chapter 4 describes the JSD development procedure, step by step. A brief example is used to illustrate the content and method of each step. The purpose of this chapter is to give a framework of the development procedure rather than to explore detailed considerations and questions arising at each step: that is the purpose of the second part of the book.

1

Model and function

1.1 FUNCTIONAL SPECIFICATIONS

Traditionally, the starting point in system development is the functional requirement. The developer begins by establishing the system's function, determining what the system is to do and what outputs it is to produce. The essential content of the system specification is the statement of system function.

This tradition has deep roots. In the earliest days of electronic computing, the computer was regarded as a machine which could be commanded, instructed, or ordered, to perform certain elementary functions. Thus we spoke of an 'add command', or a 'move instruction', or a 'multiply order'. A program or a system consisted of a set of these commands, commanding the computer to perform elementary functions in a pattern which amounted to the performance of larger, composite, functions. A suitable pattern of add, subtract, multiply, divide and move commands amounted to the calculation of gross pay for an employee in a payroll system. The invention of the subroutine in 1949 allowed larger functions to be conveniently specified in terms of smaller functions. It was then a relatively short step to view a program or system as having a single function which could be successively decomposed into smaller functions until the level of the already available machine functions was reached.

There is an obvious and attractive common sense about this. The purpose of the system is to do something, to perform some function, and that provides an apparently natural starting point for development; not merely natural, but convenient too, since the activity of decomposition, detailing, or refinement, can then lead to the final product of development, the finished system.

This is essentially the view on which 'functional decomposition' is founded. The system function is organized as a hierarchy of functional procedures, and development consists of elaborating this hierarchy from the top downwards. It is also the view underlying 'structured systems analysis and specification' (see, for example, Tom de Marco, *Structured Analysis and System Specification*, Yourdon, and Chris Gane and Trish Sarson, *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall). In that approach, the system function is organized as a hierarchy of processes connected by data flows: each process performs a part of the system function, and is itself decomposed into a set of processes connected by data flows, each one performing a sub-part of the function. But whether the medium is procedures or processes, subroutine calls or data flows, the message is function.

1.2 MODELLING REALITY

JSD relegates consideration of system function to a later step in development, and promotes in its place the activity of modelling the real world. The developer begins by creating a model of the reality with which the system is concerned, the reality which furnishes its subject matter, about which it computes.

Every computer system is concerned with a real world, a reality, outside itself. A telephone switching system is concerned with telephone subscribers, telephone handsets, dialling, conversations, conference calls. A payroll system is concerned with employees, the work they do, the pay they earn, the tax they must pay, the holidays they are entitled to. A process control system for a chemical plant is concerned with the vessels, pipes and valves of the plant, the flow of liquids and gases, the temperatures and pressures at the various points in the plant. A sales order processing system is concerned with customers, the orders they place, the products they order, the deliveries they receive, the payments they make.

It is a fundamental principle of JSD that the developer must begin by modelling this reality, and only then go on to consider in full detail the functions which the system is to perform. The system itself is regarded as a kind of simulation of the real world; as the real world goes about its business, the system goes about simulating that business, replicating within itself what is happening in the real world outside. The functions of the system are built upon this simulation; in JSD they are explicitly added in a later development step.

In JSD, we use the word 'model' to mean a model of a reality outside the computer system which is being developed. There is some scope for confusion here, because the same word is used by other writers with other meanings. Some writers use the word 'model' to denote a somewhat abstract description of the system itself or of its function. When they speak of 'modelling' they mean describing the system function in 'logical' rather than 'physical' terms, describing what the system does without giving detail of how it does it. Other writers speak of 'modelling the system' in terms of its performance