ACTION DIAGRAMS

Clearly Structured Program Design

JAMES MARTIN

CARMA McCLURE

DIAGRAMS
Structured Design

JAMES MARTIN
CARMA McCLURE

PRENTICE-HALL, INC. Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

MARTIN, JAMES (date)
Action diagrams.

Bibliography: p.
Includes index.
1. Structured programming. I. McClure, Carma L.
II. Title.
QA76.6.M3612 1985 001.64'2 85-3579
ISBN 0-13-003302-2

Editorial/production supervision: *Kathryn Gollin Marshak*
Jacket design: *Whitman Studios, Inc.*
Manufacturing buyer: *Gordon Osbourne*

Action Diagrams: Clearly Structured Program Design
James Martin and Carma McClure

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-003302-2 01

PRENTICE-HALL INTERNATIONAL (UK) LIMITED *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL CANADA INC., *Toronto*
PRENTICE-HALL HISPANOAMERICANA, S.A., *Mexico*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
EDITORA PRENTICE-HALL DO BRASIL, LTDA., *Rio de Janeiro*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

ACTION DIAGRAMS

A [illegible] BOOK

Carma McClure

ACTION Clearly Program

PREFACE

Action diagrams provide a technique that is designed to be as *user friendly* as possible for creating system structures and programs. These diagrams can be used by end users as well as systems analysts and programmers to design program logic. Because they represent program logic in a simple, easy-to-understand graphic format, they help end users understand complex logic and get it right. Action diagrams can be used to represent both high-level overviews of systems and detailed program logic. There is no need to switch diagramming techniques midstream in the design process as has been the case with earlier structured design methods. Being a top-to-bottom design tool, action diagrams support functional decomposition step by step all the way down to program code level.

Action diagrams can be used to represent the program logic for fourth-generation-language programs (and, of course, for third-generation-language programs). Although fourth-generation languages such as FOCUS, NOMAD, RAMIS, MANTIS IDEAL, and NATURAL can greatly simplify programs, fourth-generation-language programs can become subtly complex and error-ridden. Often, there are mistakes made in the use of loops, selection structures, and case structures. For this reason, it is highly desirable first to sketch out the logic of a fourth-generation-language program with action diagrams. This can help to make errors more apparent.

Action diagrams can be *data-base oriented*. Besides being able to represent all the basic *structured* control constructs—sequence, selection, case, repetition—they can also represent data-base actions. The action diagram user can draw simple data accesses—CREATE, READ, UPDATE, DELETE—performed against one instance of one record type and compound data accesses—SORT, SELECT, SEARCH, JOIN, and the like—performed against multiple instances of one or multiple record types. Most of the data-base actions in traditional data processing are simple, but as relational data bases and nonprocedural fourth-generation languages spread, compound data-base actions will become more common.

Action diagrams can be drawn manually, but much benefit derives from using an automated action diagram editor. It can help bring the power of the computer to bear on the difficult task of program design.

This book describes action diagrams. It defines the components of action diagrams and gives many examples of them. The book describes an action diagram editor for personal computers.

The concepts and techniques presented in this book are simple but powerful. They are applicable to large and small systems and programs on large and small computers. Regardless of the computer environment or the program size, the objective of action diagramming is the same: to create correct programs that are easy to maintain.

As computers continue to drop in cost, more and more end users will try their hand at writing software programs. Many have no help other than programming-language manuals. They learn the techniques of program design and coding by trial and error. As a result, they often write incorrect, unstructured programs that are never properly verified and are impossible to maintain.

To expect programs to be correct and complete without thinking out the problem before writing code is as ridiculous as building a skyscraper with plans no more rigorous than those for a garden shed. All but the simplest programs need to be designed before they are coded. All computer users need a design method that is straightforward and simple to use. Action diagrams meet this need.

If either of us were a DP executive managing a large army of analysts and programmers, we would dictate that they all use action diagrams with common action diagramming software, because then they would all speak the same clear, well-structured language. End-users involved with computing would similarly be taught to use action diagrams for specifying systems. They would understand each other's designs and be able to maintain them. A high degree of clarity would be imposed upon their thinking, and they would obtain results faster.

James Martin

Carma McClure

CONTENTS

1 DIAGRAMS AND CLEAR THINKING

MAKING A MESS

One of the problems with computing is that it is so easy to make a mess.

Even seemingly simple designs quickly grow messy. Unless we use clean, well-structured design techniques, we can easily make mistakes. When programs grow large, we sometimes have difficulty understanding our own code. If we have difficulty with *our own* code, it is much worse attempting to understand *someone else's* code. The maintenance of systems is expensive and error-prone because of this difficulty.

A LANGUAGE FOR CLEAR THINKING

Complex structures and logic can be made much easier to understand if good diagrams are used. It is said that a diagram is worth a thousand words; but in describing program structures, a thousand words can be thoroughly confusing, whereas a good diagram can reveal the structure with immediate clarity. Good diagrams are a language for clear thinking.

Since the earliest days of computing, diagrams have played an important role in representing systems and developing programs. However, the types of diagrams used have changed. In fact, we can trace the evolution of programming methodologies by noting the changes in diagramming techniques.

In the 1950s and 1960s, flowcharts were used to plan out detailed and complicated program logic. In the 1970s, structured techniques became widespread, and with these, structured diagramming techniques such as structure charts were used.

Diagramming techniques are still evolving. When we examine techniques in common use today, we see that many of them have serious deficiencies. Flowcharts have fallen out of favor because they can give neither a high-level nor a structured view of a program. Some of the early structured diagramming techniques need to be replaced because they cannot represent all the basic structured control constructs, such as sequence, selection, and repetition. Because of this, they are not good tools for the automation of programming or the creation of programs with a computer ''workbench''. environment.

TRENDS THAT AFFECT DIAGRAMMING

Today there are several important trends that affect our requirements for diagramming techniques:

- End users are becoming involved to an increasing extent in creating their own systems. A variety of end-user languages are in use for this. Diagrams that are easy to teach to end users (as opposed to DP professionals) are needed.
- The slowness of DP design and programming has become a major concern. Large improvements are needed in the productivity of system building. These will be achieved by giving systems analysts, programmers, and end users computerized tools. Diagramming techniques that are efficient with these tools are needed.
- Fourth-generation languages are coming into widespread use because results are obtained with them more quickly than with languages such as COBOL or PL/I. Diagramming techniques that link directly to fourth-generation languages are needed.
- Perhaps the most important change in the job of systems analysts and programmers is the use of computer-aided design (CAD). Designs, often involving complex logic and data structures, are created, used, and modified at the screen of a computer. The computer provides as much help as possible in creating the design, verifying it to eliminate errors, and generating executable code from it. Diagramming techniques will be a vital part of the CAD software.
- Maintenance (the modification of previously written code) is becoming an increasing problem, not only because of its cost, but also because it is preventing systems from being changed when they should be changed. This inability to change critical systems can do severe financial harm to corporations. Clear, well-structured diagrams that can be changed on a computer screen should be linked to all code to aid in the maintenance of that code.

ACTION DIAGRAMS

Action diagrams were created with these concerns in mind. They are simple and clear. They appear so obvious in their structure that people tend to ask, ''Why weren't they invented twenty years ago?'' We believe that they are the simplest and best method of drawing the structures of structured programs.

Experimentation with end users was done as the diagramming technique evolved. Non-DP professionals learn the diagrams quickly and find them easy to use.

Action diagrams are easy to edit on the screen of a personal computer.

Unlike other diagrams in common use, action diagrams can represent a high-level overview of a system and progressively decompose it until executable code is reached.

Unlike most types of diagrams in common use, action diagrams can represent all of the basic constructs of structured programming.

Action diagrams work well with fourth-generation languages. If the authors of this book were teaching a sound fourth-generation language to end users, they would start by teaching action diagrams and the control structures they represent, then fit the code of the language to the action diagrams. The material in this book ought to be basic training in information centers.

With an action diagram editor, programs can be created quickly with far fewer errors than in conventional programming. The programs can be understood and changed easily.

END-USER INVOLVEMENT

It is essential to involve end users in the software development process. Some analysts and programmers prefer to work in isolation without end-user interference or discussion. This is dangerous because it is likely to result in software that does not meet user needs.

Increasingly, some end users are developing their own software with user-friendly fourth-generation languages. Where users do not write their own programs, they should sketch their needs and work hand in hand with an analyst (perhaps from an information center) who develops the software for them. User-driven computing is a vitally important trend for enabling users to get their problems solved with computers.

Diagramming techniques are an essential part of any basic course on computing. As end users become more involved with system design and fourth-generation languages, they should be taught diagramming techniques. We suggest that they be taught action diagrams because these have been designed to be user friendly, easy to learn, easy to use, tailored for use with fourth-generation languages, and easy to manipulate on a personal computer.

BENEFITS OF CLEAR DIAGRAMS

A standard computerized way of representing system and program design enhances team communication and enables management controls. Action diagrams combine graphic and narrative notations to increase understanding. Graphics are especially useful because they tend to be less ambiguous than a narrative description. Also, because they tend to be more concise, graphics can be drawn in much less time than it would take to write a narrative document containing the same amount of information.

When a program is modified, clear diagrams are an essential aid to maintenance. They make it possible for a new programmer to understand how an existing program works and to design changes to that program. When a change is made, it often affects other parts of the program. Clear diagrams of the program structure help maintenance programmers understand the consequential effects of changes they make.

When looking for program bugs, clear diagrams are a highly valuable tool for understanding how the program works and tracking down what might be wrong.

Architects, surveyors, and designers of machine parts have *formal* diagramming techniques that they must follow. Systems analysts and program designers have even a greater need for clear diagrams because their tasks are more complex and because the work of different people must interlock in intricate ways. In the past, however, there has been less formality in programming. This has made systems much more difficult to maintain and change.

For small, one-person projects, an action diagram editor running on a personal computer helps to clarify and speed up design and programming. For large projects, standards are needed for communication among many developers. The larger the project, the greater the need for precision in diagramming. It is impossible for the members of a large project to understand in detail the work of others. Instead, each team member should be familiar with an overview of the system and see where his component fits into it. He should be able to develop his component with as little ongoing interchange with the rest of the team as possible if he has clear, precisely defined and diagrammed interfaces with the work of the others. When one programmer changes his design, it should affect the designs of other programmers as little as possible. To achieve this requires formalized techniques for representing the system design. Action diagrams meet this need.

Certain other types of diagrams are useful for conceptualizing other aspects of complex systems design. As discussed in Chapter 9, these other diagrams, if drawn with appropriate standards, can be *automatically* converted to action diagrams and then to code, on the screen of a personal computer.

LANGUAGES CHANGE OUR THINKING PROCESSES

Philosophers have often described how the language we use for thinking affects what we are capable of thinking. When the world had only Roman numerals, ordinary people could not multiply or divide. When Arabic numbers became widely used, the capability to multiply and divide spread. Diagrams for drawing computer processes are a form of language. With them we can express more complex processes than we can by using English. The designer thinks in diagrams, conceptualizes systems with the aid of diagrams, and refines his design by manipulating the diagrams. The form of a diagram has a direct effect on this process. If the diagram cannot express repetition, sequence, selection, conditions, data-base operations, or parallelism (as is true with some popular types of diagrams for systems analysis), this, like Roman numerals, tends to limit the thinking processes of the designer or user.

A skilled system designer should be fluent with several diagram types that help conceptualize different aspects of systems. Different types of diagrams of processes for which program code will be created should be *automatically* convertible to action diagrams.

The authors have discussed the many types of diagrams in use and have compared action diagrams with alternate techniques in their book *Diagramming Techniques for Analysts and Programmers* [1].

It is desirable for a corporation to standardize diagramming techniques and make this a basic part of its program and system documentation. Having such a corporate standard aids communication among designers and programmers and helps end users to achieve better communication with DP. Without such a corporate standard, maintenance of programs will be unnecessarily difficult. Today the diagramming standard should be the basis of diagramming software. This speeds up the design process, aids communication among designers, and enforces correct use of the standard technique.

We believe that action diagrams should be an essential component of such a standard [2].

REFERENCES

1. James Martin and Carma McClure, *Diagramming Techniques for Analysts and Programmers* (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985).
2. James Martin, *Recommended Diagramming Standards for Computing*, Savant Research Report (Carnforth, Lancs., England: Savant, 1984).

2 ACTION DIAGRAM BRACKETS

A BUILDING ANALOGY

If we were going to build a one-room mud hut, we would not need to do much planning before construction. Only a few materials and tools would be needed. The construction process would involve a few simple steps done by one person. If a mistake was made or a change was necessary, the hut would be easy to modify or rebuild.

At the other extreme, if we were going to build a skyscraper, we would need a detailed architectural plan. Creating this plan would be our first step since no professional builder would attempt such a project without a sound plan. Otherwise, how would we determine what building materials were needed, how many workers to hire, or what jobs to assign to them? If the plan were incomplete or incorrect, the cost to change the building or to modify the construction schedule could cause serious financial problems, possible unsafe conditions, and expensive legal consequences.

Building a single-family house falls somewhere in the middle. An experienced builder could probably construct it without a blueprint, but in most cases this would be inefficient and sometimes foolhardy. By skipping the design phase, the builder could begin actual construction sooner. However, the whole project might take longer to complete. Without a sound design, the house could be of poorer quality because its structure and construction materials would be chosen on the basis of availability rather than quality considerations. Also, because the buyer would not have been given an opportunity to review the plan and perhaps modify his requirements before construction began, he might be less satisfied with the outcome. He might demand many costly changes that would greatly reduce the builder's profit.

We can draw a strong analogy between constructing a building and constructing computer programs. It is as important to design a program as it is to design a building.

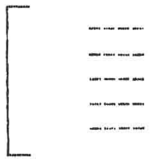
DESIGNING WITH ACTION DIAGRAMS

Diagrams are the language of design. Appropriate diagrams offer a concise, unambiguous way of describing a program or system of programs. The choice of diagramming technique has had a major effect on the efficiency of the designer and the quality of his design [1].

Action diagrams provide a natural way to draw a high-level overview of a program structure as well as a detailed view of the program logic. They employ the lessons of structured programs to make the design as clean and well built as possible. They build on the diagramming techniques of the past, discarding what has been shown to be ineffective but keeping the constructs that support structured programming. All the basic concepts of a well-structured program, such as modularization, hierarchical organization, functional decomposition, and structured control constructs, have been included in action diagrams. Regardless of whether an end user or a professional programmer is creating a program, the objective should be a structured program. Structured programs are easier to build, test, debug, and maintain.

BRACKETS

The basic building block of an action diagram is a bracket:



A bracket encloses a set of actions. An action can be a high-level function, a procedure, an operation, a program, a program subroutine, or, dropping down into detail, an individual line of program code.

The following bracket shows high-level functions:

```
┌─ SERVICE A POLICY
│
│     APPORTION REMITTANCE
│     ENDORSE POLICY
│     ASSIGN POLICY
│     ALLOCATE BONUS
└─
```