Paradise Lost:
Almost *Nobody* Knows What's *Really* Happening Inside a Modern Software Application

In the 1980s, with the advent of interactive software such as Macintosh and Windows, and with widespread adoption of other technologies such as large-capacity hard disks, business computing moved from protected vaults to desktops. As computers became more accessible and user-friendly to the people who benefited from them, the interactive software programs that enabled this migration became more obscure and incomprehensible to the programmers who built them.

Contents

Input-Process-Output

A Brief History of Mechanized File Processing

Computers came into widespread use as business tools in the late 1950s. In those days data files were held in stacks of punched cards. Each card carried a small amount of information, up to 80 letters or numbers. One card might carry the basic name-address information for one customer, for example, or the information for one line item of a sales transaction.

Punched cards have to be neatly stacked in order for processing of the cards to be mechanized. You process one card at a time: you draw the card at the beginning of the *input* stack, put it somewhere to work with it, then when you are done you put it at the end of the *output* stack. You repeat the process until the input stack is empty.

(After a few years magnetic tapes replaced punched cards. Tapes also must be processed sequentially, so the processing methods remained unchanged. I'll continue to refer to cards, but this all applies to tapes as well.)

Figure 1 shows a schematic diagram of this process.

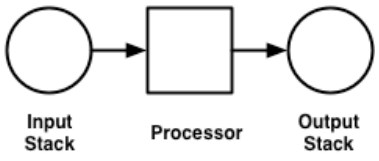

Figure 1
Schematic Diagram of a Sequential File Process

Figure 2 shows a *flow chart* that describes, step by step, what actually happens inside this basic sequential processing program.

The flow chart was the principal intellectual tool used by people designing sequential file processing programs, for two reasons: it helped them to *think* about the programs, and it was the principal medium for *documenting* the programs. You can treat the flow chart as a game board; start at the top, follow an arrow and, when you arrive at a box, perform the action named in the box. Then follow the arrow leaving the box. In the case of a diamond-shaped question box, choose the exit arrow that is labeled with the correct answer.

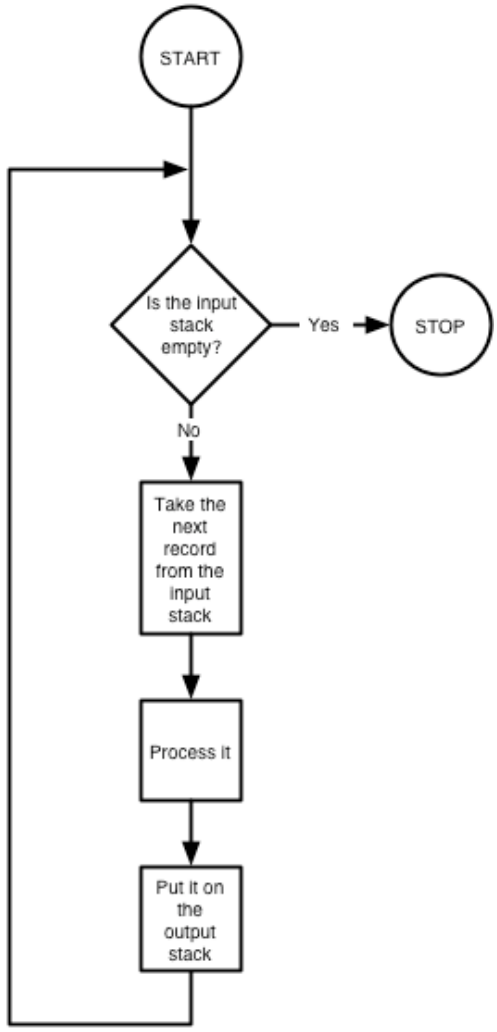

Figure 2
The Flow Chart for a Sequential File Processing Program

A program description in the form Do A, then Do B, then Do C, etc. is called a *procedural* description. We will see later that a procedural description is not necessarily the only, or indeed the most useful, way to describe how some programs work.

The Sort-Merge Processing Paradigm

Until the world changed (see the next section) the Sort-Merge paradigm was how data processing was done. *Magnetic-tape computers spent all their time sorting and merging sequential files on magnetic tapes.*

A stack of cards is sorted using some group of characters stored on every card, such as customer number. When we have a stack of cards, all representing customers and sorted in some appropriate way such as by customer number, we are entitled to call this stack a *customer file*.

Let's say we are wholesalers and we want to perform an end-of-month billing run in order to send out bills for the month's sales transactions. We perform a *merge*, in which we match up two input files, a customer file and a transaction file, and produce one output file (in this case a stack of printed invoices). Figure 3 is the schematic diagram of an invoicing merge.

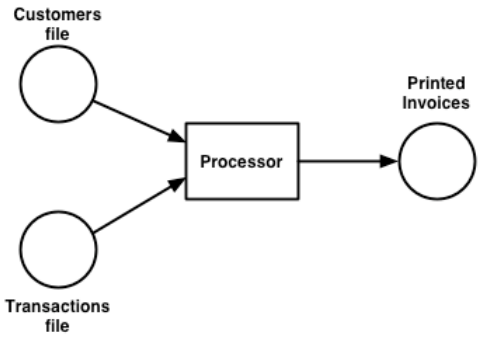

Figure 3
An invoicing merge

Early processors had very limited memory and could access the data in only one card (or magnetic tape record) in each input file at a time. Therefore we had to sort both the customer file and the transaction file by customer number before performing the merge. Sorting both files by customer number guarantees that, in one pass over the files, all the transactions for every customer will be together and can be made to show up at the same time the customer's customer card shows up. Thus all the data will be in place to print the invoice.

The invoicing program is an elaboration of the basic sequential processing program shown above in Figure 2. Figure 4 below shows its flow chart. (Bug alert: the design of this program assumes that for any customer number appearing in a transaction record there is a corresponding customer record in the customer file.)

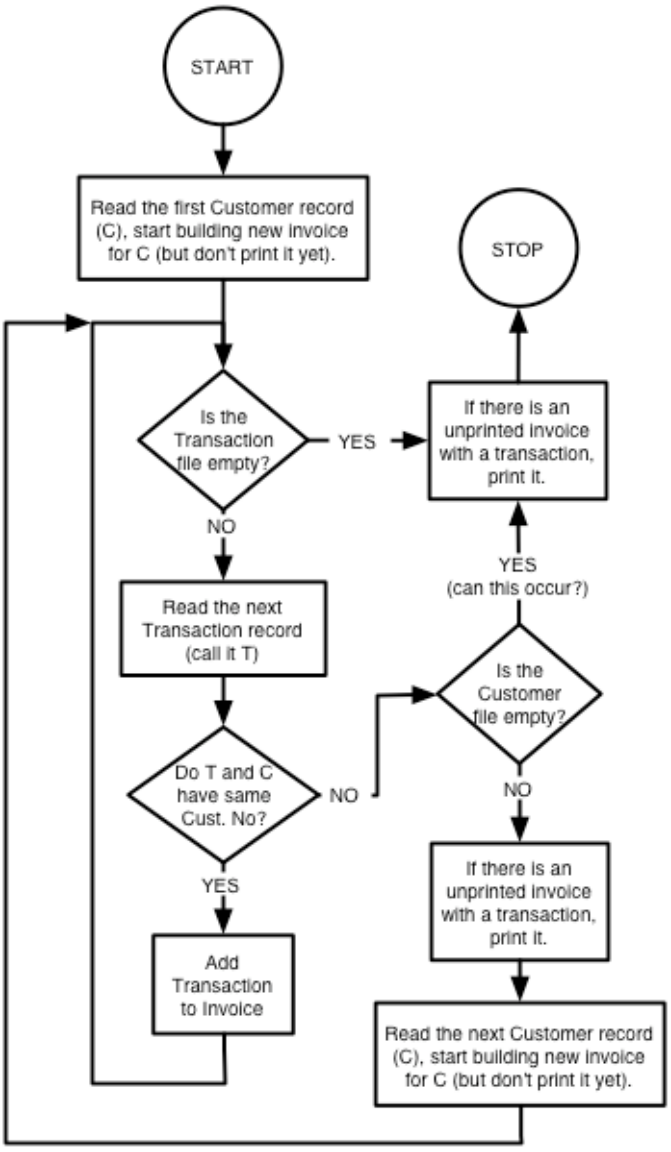

Figure 4
Flowchart For the Invoicing Merge Program

The flowchart is a roadmap that shows how to navigate through the steps of a program. *It is important to recognize what a powerful conceptual tool the flow chart is*; part of the reason for this power is that these programs have two simple properties.

1. All parts of the program are connected. The road map describes one country, not multiple countries separated by oceans.
2. There is a beginning and an end; you know where to start and you know you have reached the end when the road map takes you there.

Event-Driven

Then the World Changed

After sequential file processing had become well entrenched in the 1950s and 1960s, two revolutionary things happened. Large magnetic disks became the preferred storage medium—this eliminated any necessity for sequential processing—and (in the 1980s) *interactive* applications, initially popularized by the Macintosh and then Windows, became the norm in business.

What interactive programs do internally bears no resemblance to sequential file processing. Interactive programs are *event-driven*; an interactive program contains little pieces of program called *event listeners*, each one listening for a particular event such as a keystroke, movement of the mouse, or a mouse click. Each event listener does nothing until its particular event occurs; then it come to life and performs some small task.

Here is how the typical sequential file processing program differs from the typical interactive program.

The type of program	What it does	How it is structured	Its rhythm
Sequential file processing program.	Repetitively processes all records of one or more sequential files from beginning to end.	A loop. It keeps repeating until it runs out of records.	Start, run continuously to the end, then stop.
Graphical, interactive, event-driven application.	Edits a file or database document, a little bit at a time, in response to small events (e.g., keystrokes and mouse clicks) received from the user.	A lot of loosely connected pieces. Each event listener triggers a specific modification to the document. Each little event processor has a beginning and an end, but the overall program doesn't.	Inactivity, then a burst of activity in response to an event, then inactivity again.

The repetitive, sequential character of the sequential file processing program disappeared as the dominant business information-processing paradigm.

Event-driven programs don't have the two important properties—contiguity and distinct beginning and end—that sequential file processing programs do. The flow chart as a powerful intellectual tool for conceptualizing and documenting programs has become useless, and nothing has taken its place.

The invention of *object-oriented programming* has helped to rationalize the internal structure of event-driven programs; object-oriented programming did this by creating a set of not-easy-to-describe *abstractions* out of which event-driven programs are now built.

The flowchart of Figure 4 is something a naïve, intelligent student can study and understand in an hour; at the end of the hour she will have a degree of confidence that she understands what happens within a sequential file processing program. If necessary, she can simulate the operation of the program by preparing a small deck of customer cards and a small deck of transaction cards, pretend she is the processor, and play out the program's behavior.

This exercise is simply not possible for learning the behavior of an event-driven program because an external agent (the user) triggers the little event processors in no predictable order. Acquiring an understanding of all the possible paths the program can traverse has become a much more complex exercise.

Building high-quality event-driven programs is also harder. Event-driven programs must defend themselves against the entry of incorrect data and of other incorrect event sequences; this necessity enlarges the number of test cases that must be considered compared to programs that don't have to deal directly with unpredictable operators.

Most programmers are too young to realize that their discipline used to be much simpler; they accept obscurity and complexity as normal. They have learned to skirt around some of the difficulties by living in the world of the abstractions provided by object-oriented programming. The price programmers (and wannabe programmers) are paying is that they no longer are able to think in concrete step-by-step terms about *what really happens* inside an event-driven program, and there is no longer any brain-amplifying tool such as the flow chart that they can use to wrap their minds around the gestalt of an event-driven program.

The economic and human consequences of this increase in obscurity and complexity have developed slowly over decades and are not obvious to those without an historical perspective. Today, to be hired as a programmer in a business information-technology organization, the candidate must have at least a Bachelor's degree in Computer Science. In the 1950s, when punched card technology was dominant and

programs were created by wiring plugboards, a business data-processing operation comprised a flock of different types of punched-card processing machines, each with a specific, narrow function. IBM conducted courses that prepared people to be proficient in the programming and operation of any one of these machines in a week or so; a few of the more complex machines required several weeks. It was possible to enter the data-processing workforce right out of high school.

This is Progress?

The situation programmers find themselves in reminds me of a 1946 paper reporting experimental results in the psychology of learning. The following figure appeared in the paper. These are multiple pictures of the same cat, obtaining food on successive occasions.

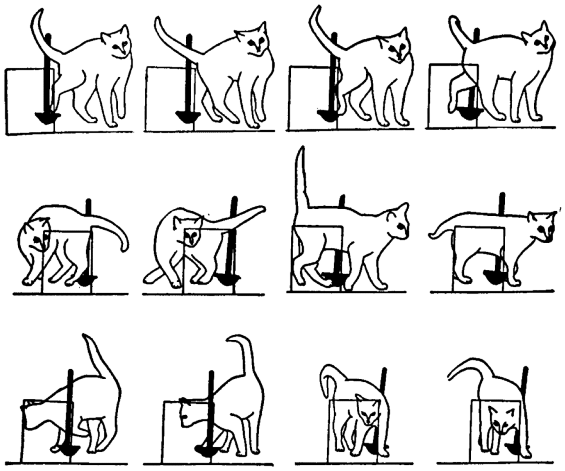

Figure 5
How this cat got its meals.

In the experiment each cat learned to obtain food by pushing a pole that stuck out of the floor of its box. The particular cat made famous by this picture first succeeded in obtaining food by backing into the pole, and that's how it got its food from then on.

The cats in the experiment repeated what produced results without understanding how their world worked. Programmers are in the same fix.

Experimental Subject	Problem	Solution
Cat	Produce a meal	Push stick with butt
Programmer	Produce an event-driven program	Push around object-oriented abstractions