The Art of Solving Problems with Computers

Howard S Modell
The University of Phoenix

Copyright 2002, Howard S. Modell

Introduction.

Programming, Music, Art[*], and Magic are surprisingly related activities. They all involve the use of creativity and insight, although neither of those traits is a requirement. Just as almost anyone can learn to play an instrument[†] so, too, almost anyone can learn to write computer programs. It doesn't mean that you can do the activity well; *talent* still counts for something, but it doesn't take talent to produce notes on an instrument, or to paint-by-the-numbers, or to design and code simple programs. As for what Programming and Magic have in common, for now all I will say is that:

- **The Naming Of Names** is important;
- Mastery of **Rituals** is important;
- the use of **Diagrams** can be very helpful;
- and **Repeatability** is a Virtue. I will discuss this more later.

The abilities required to be able to design computer programs include: being able to delineate a procedure for accomplishing the task "by hand"; being able to recognize the different kinds of data involved in the task, and to determine ways in which that data can be organized to make it easier to work with; and lastly (and actually the easiest part) to be able to model the procedure into the terms and syntax of some programming language [PL].

Of the "required abilities", the most critical one is the first one, to be able to describe the procedure required for performing the task involved. It is essential that, before you can even try to "tell" the computer how to solve a problem that you have to know how to solve it. It may well be that the solution you know is inconvenient or even

---

[*] By "Art" I mean graphical arts like Painting and Sculpting.
[†] so long as one is neither totally tone-deaf nor unable to sustain a 'beat'

impractical for a human to do[*] but as long as it would solve the problem, it can be considered a valid procedure..

That solution description can be set down in different ways, each one with its own strengths and weaknesses. One can use plain English, or an "Input-Process-Output" (IPO) chart, or a flowchart, or pseudo-code (to name just 4 ways). There's also nothing to stop one from using multiple methods, starting from a general description and refining it in different ways as one goes along. Whatever methodology one uses, whatever notation one works with, the important thing is to account for all the steps involved. You must always keep in mind that computers are functional "morons". A computer does not *know* anything; a computer does only what you can tell it to do, and it will do it exactly the way that you tell it to do it, it will repeat any mistake perfectly and repeat ably, and it won't stop what it is doing unless and until you tell it to stop.

A Simple Problem

Let's look at a simple problem and see where it leads us.

***The Problem*: Determine if a specified number 'N' is a 'Prime Number'**

First, we need to think about the problem (analyze it) and convince ourselves that we understand it. We ask ourselves, "what is a 'Prime Number'?" In Mathematics, a "Prime Number" is defined as a positive integer (i.e., a whole number) with the property that its only whole divisors are itself and 1. For "J" to be a "whole divisor of N" means that when J divides N, the remainder-after-division is 0[†]. For example, when 3 divides 10, the result is 3 with a remainder of 2.

The Analysis

That sounds like a fairly straightforward definition. We can visualize a "Prime-ness test" procedure along the lines of:

---

[*] just because you could count to a googol -- a BIG number, represented by a 1 followed by 100 zeros -- by ones doesn't mean that you can do so in useful time, before you become old and feeble

[†] Another way of saying this is "N modulus J is 0"

For any given number N, we can try dividing it by all the successive integers from 1 up-to-and-including N itself, looking at the remainders-after-division. If any of the divisors other than 1 and N produce a remainder of 0, then N is not "Prime"; otherwise it is.

We can rewrite this as a "program-like" procedure this way:

1. accept an integer as input, store it as the value of a variable "N"
2. compute the consecutive integers from 1 up to N (inclusive), and …
3. … for each integer, (call it "J"), calculate the value of *N modulus J*.
4. If the result of (3) is 0 and J is neither 1 nor N, then tell the user that "N is not a Prime Number".
5. If the result of (4) is non-zero, then repeat steps 3 through 5 until you've finally tested N with itself. If you get to the point of calculating *N modulus N*, then you're done and you can tell the user "N is a Prime Number".

One thing to be very careful about during the design process is "stay general". Do **not** design your solution in terms of any programming language. "Language shapes Thought": the vocabulary we use, the concepts we can verbalize, *the concepts that occur to us*, the way we structure statements can have a "freezing" effect on the design process, making it difficult if not impossible to see a useful procedure or technique just because the particular language you're thinking in doesn't inspire thoughts of that particular solution. Be prepared to consider an algorithm or technique that worked for a similar situation previously, even if the PL used at the time isn't one you might be considering for this situation.

IPO Chart

Inputs	Processing	Output
N (integer to test for "Primeness"	For each integer "J" from 1 to N, calculate N%J. • If result is 0 and J!=1 and J!=N, then N is NOT a Prime number • If no J divides N except 1 and N, then N is a	"N is NOT Prime"

	Prime number	"N is Prime"

FLOWCHART

We can try creating a flowchart to diagram the logic and analyze it that way. A flowchart is a diagram where directed lines (i.e., lines with direction .. "arrows") connect different symbols in such a way that by following the arrows we flow with the logic of the program from START to FINISH. The different geometrical shapes are used to represent parts of the procedure, each shape being used for a different kind of part (calculation, decision, cycle, etc.)

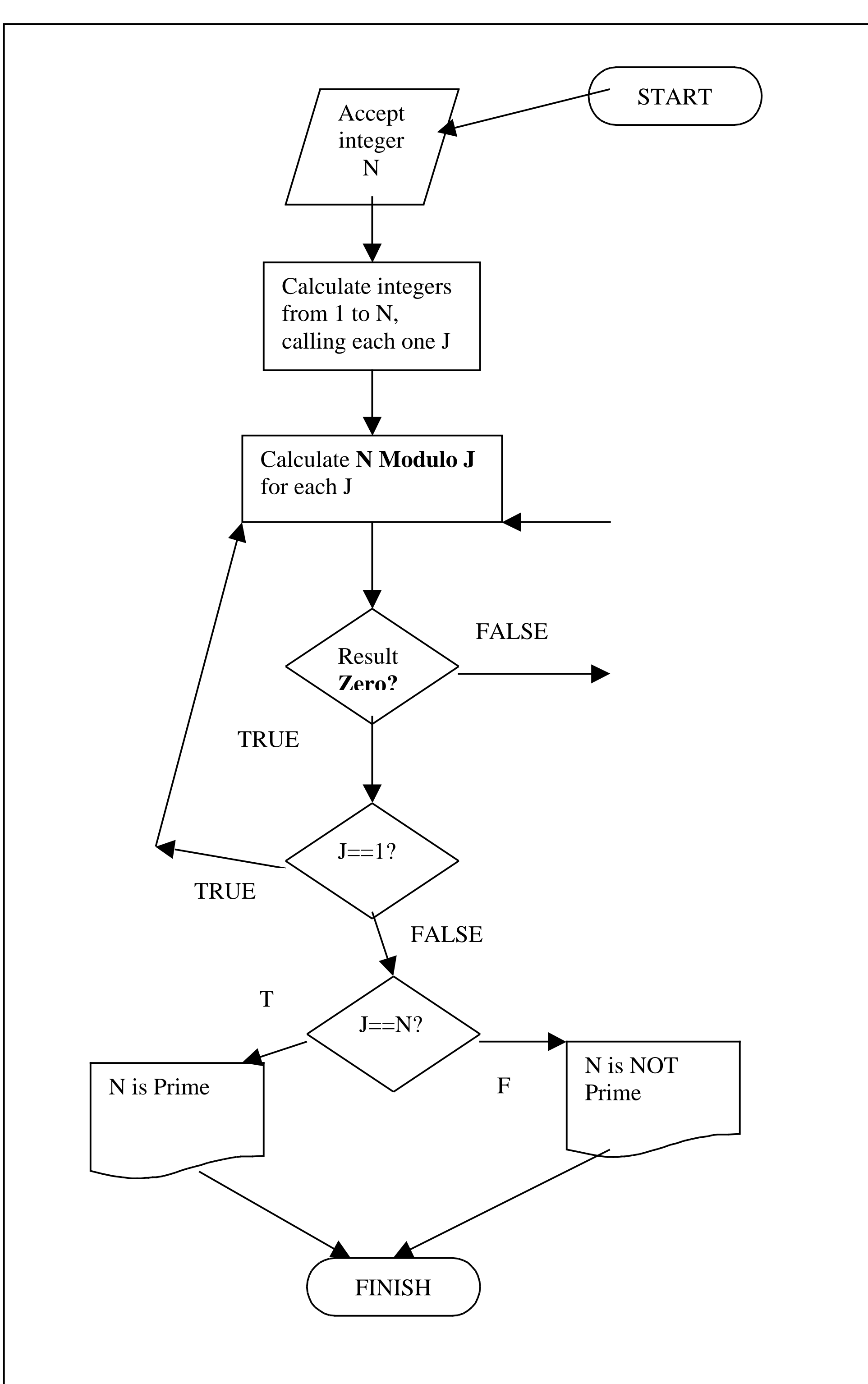
The symbols we use in flowcharts are:

	Represents a calculation
	Used to denote a decision (test) point
	Used to indicate a point where the program waits for input from user
	Used to indicate a point where the program displays information to the user
	Denotes the destination of a branch or cycle.
	Used to indicate the START and STOP points of the logic flow

Table 1. common flowcharting symbols

If we drew a flowchart for the procedure we have so far, we'd come up with something like Figure 1.

Figure 1. FLOWCHART of Simple Solution



Creating Source Code

We can try to code what we have, and we'd come up with something like this (coding in C++):

```
#include <iosteam>
using namespace std;
void main ( )
{
int N, J;
cin > N;
for (J=1; J<=N; J++) {
if (N%J == 0)  {
if (J==1) continue;
if (J != N) {
cout << "N is NOT Prime" << endl;
exit;
}
}
}
cout << "N is Prime" << endl;
}
```

Second Thoughts and Revisions

The program as written will work, but it is very much a "brute force" approach. For one thing, it tests "N modulo 1" which is always going to result in 0. So one thing we could do to make the logic a bit easier is start the set of divisors with 2. But, *ah hah!*, while 2 is a Prime number, it is the only **even** Prime number. Thus, if N%2 is 0, then N is an even number and can't be a Prime number. Therefore, we don't need to use any even integer except 2. So now we've determined that we only need to divide N by the **odd** integers greater than 1.

We could stop there, but there is one more thing we can do to make our program better still. A little thought tells us that if "x" divides N, then so does N/x. So if x does NOT divide N, then neither will N/x. So we really only need to consider the odd integers up to the square root of N*. If we haven't found a whole divisor by the time we try square-root(N)+1, then we're not going to find one at all and we can conclude that N is Prime. This gives us a slightly better program thusly:

```
#include <iostream>
using namespace std;
void main()
{
int N, J;

cin >> N;
if (N == 1) goto NotPrime;
if (N == 2) goto Prime;
if (N%2==0) goto NotPrime;
for (J=3; (J*J<=N); J++)
{
if (J%N == 0) goto NotPrime;
}
Prime:
cout << "N is Prime" << endl;
exit (0);
NotPrime:
cout << "N is NOT Prime" << endl;
exit (0);
}
```

* the "square root", S, of a number N is the value such that S*S == N. This makes is sort of a "median" divisor of N in that all the other divisors are either <S or >S.

Summary

The example in this paper is a relatively trivial problem and solution, but it does nicely illustrate some of the aspects of program design. We should never dive straight into coding. At the very least we should think about the problem we are attempting to solve, if only to make sure that we understand the problem properly and completely. Then, regardless of whether you use flowcharting, IPO charts, pseudo-code, or some other "design-capture" method, it is always better if you put your solution ideas into a form so that you can examine them, evaluating those ideas to see if your proposed solution covers the whole of the problem.

Capturing the logic of the proposed program into a document of some kind also makes it easier to check your logic (i.e., to "desk check" it). It is, by the way, perfectly permissible to solicit someone else to review your proposed solution with you. Often a fresh pair of eyes may see something where you are so familiar with your own ideas that you can miss the obvious.

Once you've determined that your proposed procedure is good enough, you can translate it into source code. The translation process should be relatively easy, but if it is too easy, then you may have let your knowledge/familiarity with the particular PL influence the design too much. For **most** solutions, the design should be as "platform-independent" as you can keep it. The only times when it is useful to design for a specific platform (hardware or software) is when the explicit advantages of that platform can be shown to outweigh the arguments for "staying general". [Certainly, programming with domain-specific languages like GPSS (for simulations), APL (mathematics), SNOBOL4 (string-processing and pattern-matching), or Prolog (formal logic) produces more elegant solutions for problems in that domain than would trying to use a general-purpose language.] Just as some instruments are easier for some people to use than they are for others, so too are some programming languages easier for some people to work in. If you spend most of your time working problems in a specific "problem domain", see if there is a PL designed to work well in that domain.

Having finally chosen a PL for implementation, if it provides facilities for building "testing" into the program itself, *make use of such features.* Features like macros, "assertions", and exception-handling may well allow you to find logic errors

faster than trying to do it by "eye". If the compiler is part of an "Integrated Development Environment", use its facilities, especially the "run-time debugger" if there is one.

The Art of Programming is no more difficult than learning to play an musical instrument is, and takes just as much Time and Effort. Like Music and Graphic Arts, program design comes more easily the more you practice it.

The Magic of Programming

As mentioned briefly above, there are aspects of Programming that have a distinct "magical" feel to them. When we engage ourselves in programming, there is often a feeling that we're trying to convince the computer to do what we want it to. In its loosest context, "Magic" can be defined as a "methodology" which combines "material components" (charts, potions, icons, etc.) and "verbal components" (proper spells and chants) in various ways to control the environment and "make things happen". A "sorcerer" ("sourcerer"?) at work knows that everything must be set up "just so" if this spell or incantation is going to accomplish its purpose.

The Naming of Names.

When composing spells, drawing charts, or even writing things in one's grimoire, a sorcerer knows that he must correctly and accurately name the things and Beings he will reference. Misnaming a "demon" can get you either the Wrong Demon or a very Angry Demon. In the context of programming, naming things correctly and accurately is also of critical importance. When programming in a language like PL/1 where keywords are not reserved, using them as variable can generate confusion very easily, as for example

```
IF IF THEN THEN ELSE ELSE;
```

Things are a little better in languages like PASCAL or C/C++ where keywords are reserved, but in all programming languages "an identifier is an identifier is an identifier". Generally, any of the identifiers created by the programmer can be used in all contexts where "identifier" is valid. Languages with "strong typing" help a little, by

adding the restriction that a variable can only be used in a context appropriate to its declared "type". But it is still the case that any integer variable can be used in an integer-context. The compiler has no way of knowing that ItemPrice1 was what you meant to use and not ItemPrice10.

Programmers are often taught that "names are better than literals". If the PL allows for it, using macros or "compile-time constants" to give names to constants is supposed to be a Good Thing. And, yes, using **ONE** instead of 1 makes for a potentially more readable and more maintainable program. The problem here, and what isn't always taught, is that names should be *meaningful* . While **ONE** is better than 1, **ItemCount_InitialValue** is better still, and less likely to cause problems down the line when the design changes and **ItemCount_InitialValue** needs to be 5. A lot of programmers are used to using **I**, **J**, **K** as loop-counter variables. All well and good, especially if those variables really are just used in the context of those loops, but it is still very easy to mistype **J** for **I** and not realize it unless you have a good debugging tool to work with.

Diagrams and Charts: The Use of Visual Aids

Every Astrologer knows that the Chart is everything. You can tell the client anything, but they'll be more likely to believe you if you have a good, flashy Chart to show them. And, charts are much easier to work with in composing a horoscope that simple columns of numbers are. Our distant ancestors know about the Usefulness of Diagrams, too. It is commonly believed that a lot of the cave paintings we've found are probably rudimentary attempts at "hunting magic": draw a picture of an animal being brought down by hunters with spears and it may cause that happening to come about the next day. Sorcerers drew pentagrams on the floor as the means to delineate where they and their assistants had to stand, and where the Thing Summoned would be caged.

In the context of programming, we have our versions of astrological charts and pentagrams. Just as a magician is ill-advised just to "dive straight into his spell" without thinking about what he needs to do first, so too should a program designer takes some time to think out his design before ever composing a line of code. The value of methods like IPO charts and flowcharts is simply that they force the program designer to attempt

to list the data and the processing logic in language-independent ways. Just as using runes and other symbolic languages can help a magician to properly compose the spell he has in mind, so too can pseudo-code help a program designer to visualize the logic of the program he has in mind.

The Importance of Ritual: **Saying the Right Things in Correct Fashion.**

Leaving aesthetics and "customer satisfaction"* aside, sorcerers knew that it wasn't just important to know What to say, but How to phrase it. Demons reacted better (it was thought) to flowery, flattering phrases than they did to commands. The ritual worked (?) when you Named the Four Compass Points (homes presumably to the Winds) in the Correct Order. Any spell had to be written following the "formula" correctly: light the candles in the Correct Order, throw just the Right Amount of Incense into the brazier at just the Right Time, and so forth, if you wanted it to have a hope of working.

Certainly, it is easy to point out the frustration that comes from simple syntax errors arising from careless behavior on our part – leaving out a semi-colon, for example, or forgetting to add a closing quotation-mark (thus sucking all of the program code following the offended string into that string). PL's have very definite syntax, constraining how things can be specified and sometimes even the order of the program elements (e.g., variables and functions must be declared before their first reference).

Repeatability **is a Virtue**

One of the places where the analogy between Magic and Programming breaks down is in the area of "repeatability". Much as they might try, magicians and alchemists had a very difficult time getting the same result twice*. Computers are blithe morons, fully capable of repeating actions forever (or until the next power outage). Even where we try to build in "randomness" into the algorithm, we can only partially succeed. PL's that provide "pseudo-random number generators" also point out what the "pseudo-"

* Priests and Shaman have long known that their followers are much easier to lead and much more content with "the show" if it is done in Latin or some other language that the audience doesn't know. What I thought was a prayer that was an essential part of a Jewish service I used to participate in turned out to be, upon looking at the translation on the opposing page, a recipe for incense.

aspect means: if you "seed" the generator with the same number, it will produce the same sequence of numbers it did the previous time.

On the other hand, even wizards knew the value of documentation. The whole point of writing down a Spell was to promote its repeatability, even if only by the wizard in question ("I did *this*, then **this**, then <u>this</u> … and ***<u>this</u>*** resulted!"). While "throw-away"[†] code is a common enough phenomenon, it is more the case that a program once written and debugged will be used again and again. Thus, a program should be written so that it is understandable, preferably in conjunction with internal documentation (comments) that describe the salient points and features of the solution.

Parametric Magic

With this goal (i.e., re-use) in mind, programmers are encouraged to enhance this future use by parameterizing their program, designing them so that certain data comes in from the outside which the program processes to decide what to do *this time*. Just as a Spell or Alchemical Recipe might have places where certain "legal substitutions"[‡] might be made to vary the outcome, so too should programs be "re-usable", able to perform the same chores with different data, perhaps in slightly varied ways.

Coda

It can not be stated strongly enough here that the preceding was intended for entertainment. No claim is being made that "Magic" has ever existed or worked outside of a theater. Computers, while their mechanisms often seem magical to the casual user, are entirely understandable, and produce results that are always repeatable (and most of the time, reliable).

---

[*] On the other hand, they *did* get <u>very</u> good at coming up with excuses why things didn't work the way they expected to.
[†] Code written to solve a problem *now* that we "*know* we won't ever need again…" (*yeah, right!*)
[‡] material spell components were often specified or selected because of certain abstract properties they had; if you were trying to levitate something, you might include a bird's

That being said, it *is* amusing to note all the similarities and parallels in the two world-views represented. Both "arts" seek to provide Control, whether it be over "spirits" and the "elements", or whether it is over mounds of numbers or useful "devices". Both arts work better when the practioner exercises their minds properly and does proper preparation rather than simply "diving in" and "just doing it". Both arts are enhanced by good documentation discipline.

If Merlin had had electronic computers available to him, what could he have accomplished?

---

wing. If you couldn't find something that the recipe called for, you could substitute something with similar properties or associations.