Problem-solving: Program Analysis and Design

(Pseudocode - Flowcharting)

6th Lecture

1. Introduction

A ***computer*** *is a useful tool for solving a great variety of problems*. To make a computer do anything (i.e. solve a problem), you have to write a computer program. The computer then executes the program, following each step mechanically, to accomplish the end goal. As we discussed previously, the *sequence of steps to be performed in order to solve a problem by the computer is known as an algorithm*. *Programs usually implement algorithms*, and others, like an operating system, implement many algorithms. *To go from an algorithm to a program is to go from an idea to a concrete thing*. Often you have to fill in **gaps** that the algorithm does not specify but is intuitively **understandable** to humans. ***Program analysis and design is the process that an organization uses to develop a program***. It is most, often an iterative process involving research, consultation, initial design, testing and redesign. Thus we can say, the purpose of program analysis and design are to solve the problem.

2. Program Analysis

In computer science, **program analysis** *is the process of automatically analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety, and liveness.* Program analysis *focuses on two major areas*: program optimization and program correctness. The first focuses on *improving the program's performance while reducing the resource usage*. The second focuses on *ensuring that the program does what it is supposed to do*. Program analysis *contributes to Software Intelligence* by providing information used in the mastering and understanding of software systems. Program

analysis *can be performed without executing the program* (static program analysis), during runtime (dynamic program analysis) or in a combination of both.

2.1. Static program analysis

In the ***context of program correctness***, static analysis *can discover vulnerabilities during the development phase of the program.* These *vulnerabilities are easier to correct than the ones found during the testing phase* since static analysis leads to the root of the vulnerability. Due to many forms of static analysis being computationally undecidable, *the mechanisms for doing it will not always terminate with the right answer* – either because they sometimes return a false negative ("no problems found" when the code does, in fact, have problems) or a false positive, or because they never return the wrong answer but sometimes never terminate. Despite their limitations, the first type of mechanism might ***reduce the number of vulnerabilities***, while the second can sometimes ***give strong assurance of the lack of a certain class of vulnerabilities***.

2.2. Dynamic program analysis

Dynamic analysis can use runtime knowledge of the program to increase the precision of the analysis, while also providing runtime protection, but it can only analyze a single execution of the problem and might degrade the program's performance due to the runtime checks.

3. Program Design

Program Design *is the phase of a computer program developed in which the hardware and software resources needed by the program are identified and the logic to be used by the program is determined*. Program Design **consists** of the *steps a programmer should do before they start coding the program in a specific language*. These steps when properly documented will make the completed program easier for other programmers to maintain in the future.

A ***characteristic*** of programs is that you and others will seek to modify your program in the future. The program's meaning is conveyed by statements, and is what the computer interprets. Humans read this part, which in virtually all languages bears a strong

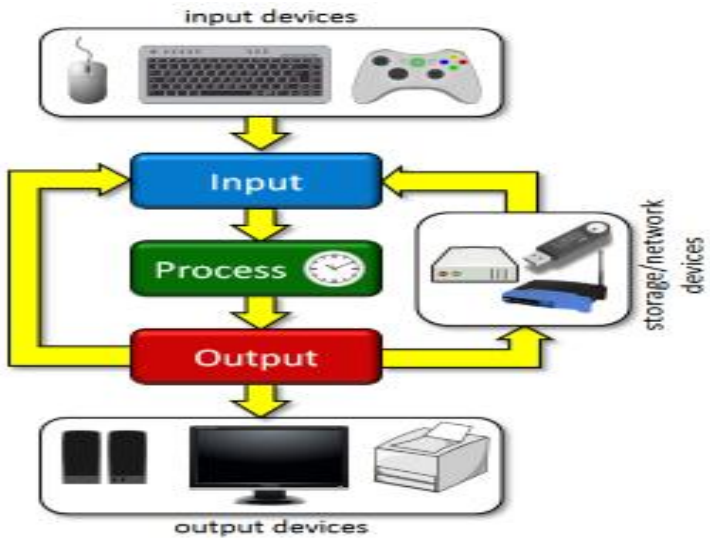
relationship to ***mathematical equations***, and also ***read comments***. *Comments are not read by the computer at all, but are there to help explain what might be expressed in a complicated way by programming language syntax*.

Activities involved in program design: There are *three broad areas of activities* that are considered during program design:

- ✓ Understanding the program
- ✓ Using design tools to create a model
- ✓ Develop test data

4. Problem Solving

Computer science is all about solving problems with computers. The problems that we want to solve *can come from any real-world problem* or *perhaps even from the abstract world*. We need to have a standard systematic approach to solving problems. Since we will be using computers to solve problems, it is important to first understand the computer's information processing model.



Consider a simple example of how the input/process/output works on a simple problem:

Example: Calculate the average grade for all students in a class.

i. Input: get all the grades … perhaps by typing them in via the keyboard.

ii. Process: add them all up and compute the average grade.

iii. Output: output the answer to the monitor.

Problem Solving *is the sequential process of analyzing information related to a given situation and generating appropriate response options*.

There are *six steps* that you should follow in order to solve a problem:

1. Understand the Problem
2. Formulate a Model
3. Develop an Algorithm
4. Write the Program
5. Test the Program
6. Evaluate the Solution

Let us now examine the six steps to problems solving within the context of the above example.

STEP 1: Understand the Problem:

The first step to solving any problem is to make sure that you understand the problem that you are trying to solve.

In this example, we will understand that the input is a bunch of grades. But we need to understand the format of the grades. Each grade might be a number from 0 to 100 or it may be a letter grade from A+ to F. If it is a number, the grade might be a whole integer like 73 or it may be a real number like 73.42. We need to understand the format of the grades in order to solve the problem. We also need to understand what the output should be. Again, there is a formatting issue. Should we output a whole or real number or a letter grade? Finally, we should understand the kind of processing that needs to be performed on the data. This leads to the next step.

STEP 2: Formulate a Model:

Now we need to understand the processing part of the problem. *Many problems break down into smaller problems that require some kind of simple mathematical computations* in order to process the data. In our example, we are going to compute the average of the incoming grades. So, we need to know the model (or formula) for computing the average of a bunch of numbers. If there is no such "formula", we need to develop one. Often, however, the problem breaks down into simple computations that we well understand.

In order to come up with a model, we need to fully understand the information available to us. Assuming that the input data is a bunch of integers or real numbers $x_1 + x_2 + \ldots + x_n$ representing a grade percentage, we can use the following computational model:

$$\text{Average1} = (x_1 + x_2 + x_3 + \ldots + x_n) /n$$

where the result will be a number from 0 to 100.

That is very straightforward (assuming that we knew the formula for computing the average of a bunch of numbers). *However, this approach will not work* if the input data is a set of letter grades like B-, C, A+, F, D-, etc.. because we cannot perform addition and division on the letters. This problem solving step must figure out a way to produce an average from such letters. Thinking is required. After some thought, we may decide to assign an integer number to the incoming letters as follows:

$A^+ = 12$	$B^+ = 9$	$C^+ = 6$	$D^+ = 3$	$F = 0$
$A = 11$	$B = 8$	$C = 5$	$D = 2$	
$A^- = 10$	$B^- = 7$	$C^- = 4$	$D^- = 1$	

After some thought, we may decide to assign an integer number to the incoming letters as follows: If we assume that these newly assigned grade numbers are $y_1, y_2, \ldots, y_n$ then we can use the following computational model:

$$\text{Average2} = (y_1 + y_2 + y_3 + \ldots + y_n) /n$$

where the result will be a number from 0 to 12.

As for the output, if we want it as a percentage, then we can use either Average1directly or use (Average2/12), depending on the input that we had originally. If we wanted a letter grade as output, then we would have to use (Average1/100*12) or (Average1*0.12) or Average2 and then map that to some kind of "lookup table" that allows us to look up a grade letter according to a number from 0 to 12. Do you understand this step in the problems solving process? It is all about figuring out how you will make use of the available data to compute an answer.

STEP 3: Develop an Algorithm:

Now that we understand the problem and have formulated a model, it is time to come up with a precise plan of what we want the computer to do. As we mentioned previously about

an algorithm, which is a *precise sequence of instructions for solving a problem*. Some of the algorithms are not necessarily in sequence. To develop an algorithm, we need to represent the instructions in some way that is understandable to a person who is trying to figure out the steps involved. Usually, two commonly used representations for an algorithm is by using either Flowcharts or Pseudocode.

A) Flowcharts

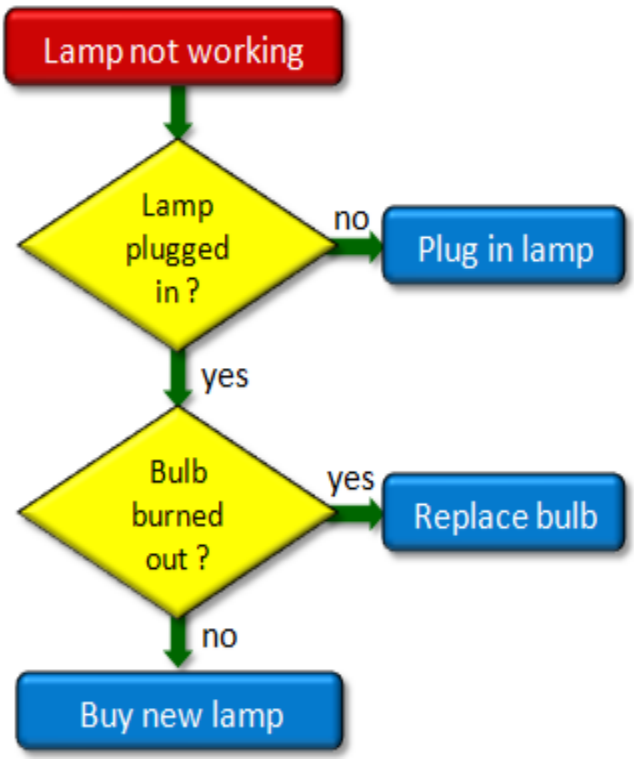
Flowcharting *is a tool developed in the computer industry, for showing the steps involved in a process.* A flowchart *is a diagram made up of boxes, diamonds and other shapes, connected by arrows - each shape represents a step in the process, and the arrows show the order in which they occur.* Flowcharting combines symbols and flowlines, to show figuratively the operation of an algorithm. *If the flowchart is too messy to draw*, try starting again, but leaving out all of the decision points and concentrating on the simplest possible course. Then the session can go back and add the decision points later. It may also be useful to start by drawing a high-level flowchart for the whole organization, with each box being a complete process that has to be filled out later.

Symbol	Name	Function
	Process	Indicates any type of internal operation inside the Processor or Memory
	input/output	Used for any Input / Output (I/O) operation. Indicates that the computer is to obtain data or output results
	Decision	Used to ask a question that can be answered in a binary format (Yes/No, True/False)
	Connector	Allows the flowchart to be drawn without intersecting lines or without a reverse flow.
	Predefined Process	Used to invoke a subroutine or an interrupt program.
	Terminal	Indicates the starting or ending of the program, process, or interrupt program.
	Flow Lines	Shows direction of flow.

General Rules for flowcharting

- All boxes of the flowchart are connected with Arrows. (Not lines)
- Flowchart symbols have an entry point on the top of the symbol with no other entry points. The exit point for all flowchart symbols is on the bottom except for the Decision symbol.
- The Decision symbol has two exit points; these can be on the sides or the bottom and one side.
- Generally, a flowchart will flow from top to bottom. However, an upward flow can be shown as long as it does not exceed 3 symbols.
- Connectors are used to connect breaks in the flowchart. Examples are:
  - ✓ From one page to another page.
  - ✓ From the bottom of the page to the top of the same page.
  - ✓ An upward flow of more then 3 symbols.
- Subroutines and Interrupt programs have their own and independent flowcharts.
- All flow charts start with a Terminal or Predefined Process (for interrupt programs or subroutines) symbol.
- All flowcharts end with a terminal or a contentious loop.

Consider the following example of solving the problem of a broken lamp. The example of Flowchart for solving this problem:



B) Pseudocode

Pseudocode *is a simple and concise sequence of English-like instructions to solve a problem.* Pseudocode *is often used as a way of describing a computer program to someone who doesn't understand how to program a computer*. *When learning to program, it is important to write Pseudocode* because it helps you clearly understand the problem that you are trying to solve. It also helps you avoid getting bogged down with syntax details (i.e., like spelling mistakes) when you write your program.

Although flowcharts can be visually appealing, Pseudocode *is often the preferred choice for algorithm development* because:

- It can be difficult to draw a flowchart neatly, especially when mistakes are made.
- Pseudocode fits more easily on a page of a paper.
- Pseudocode can be written in a way that is very close to real program code, making it easier later to write the program.
- Pseudocode takes less time to write than drawing a flowchart.

Consider the following example of solving the problem of a broken lamp. The example of Pseudocode for solving this problem:

1. IF lamp works, go to step 7.
2. Check if lamp is plugged in.
3. IF not plugged in, plug in lamp.
4. Check if bulb is burnt out.
5. IF blub is burnt, replace bulb.
6. IF lamp doesn't work buy new lamp.
7. Quit ... problem is solved.

Pseudocode will vary according to whoever writes it. That is, one person's Pseudocode is often quite different from that of another person. However, there are some common control structures (i.e., features) that appear whenever we write Pseudocode.

- **sequence**: listing instructions step by step in order (often numbered)

1. Make sure switch is turned on
2. Check if lamp is plugged in
3. Check if bulb is burned out
4. …

- **condition**: making a decision and doing one thing or something else depending on the outcome of the decision.

```
if lamp is not plugged in
   then plug it in
```

```
if bulb is burned out
  then replace bulb
otherwise buy new lamp
```

- **repetition**: repeating something a fixed number of times or until some condition occurs.

```
repeat
   get a new light bulb
   put it in the lamp
until lamp works or no more bulbs left
```

```
repeat 3 times
   unplug lamp
   plug into different socket
...
```

- **storage**: storing information for use in instructions further down the list

```
x ← a new bulb
count ← 8
```

- **jumping**: being able to jump to a specific step when needed

```
if bulb works
   then goto step 7
```

The point is that there are a variety of ways to write Pseudocode. The important thing to remember is that *your algorithm should be clearly explained with no ambiguity* as to what order your steps are performed in. Consider our previous example of finding the average of a set of *n* grades. What would the Pseudocode look like? Here is an example of what it might look like if we had the example of n numeric grades x_1, x_2,....,x_n that were input from the keyboard:

Algorithm: DisplayGrades

1. set the sum of the grade values to 0.
2. load all grades x_1, x_2, …, x_n from keyboard.
3. repeat *n* times
4. get grade x_I
5. add x_i to the sum
6. end_repeat
7. compute the average to be sum/n.
8. print the average.

STEP 4: Write the Program:

We now have to transform the algorithm from step 3 into a set of instructions that can be understood by the computer. Writing a program is often called "**writing code**" or "**implementing an algorithm**". So the code (or source code) is actually the program itself. Below is a program (written in processing) that implements our algorithm for finding the average of a set of grades. Notice that the code looks quite similar in structure, however, the processing code is less readable and seems somewhat more mathematical:

Pseudocode	Processing code (i.e., program)

1. set the sum of the grade values to 0.
2. load all grades x_1 … x_n from file.
3. **repeat n times** {
4. get grade x_i
5. add x_i to the sum
}
6. compute the average to be sum / n.
7. print the average.
	`int sum = 0;`
	`byte[] x = loadBytes("numbers");`
	`for (int i=0; i<x.length; i++)`
	`sum = sum + x[i];`
	
	`int avg = sum / x.length;`
	`print(avg);`

For now, we will not discuss the details of how to produce the above source code. In fact, the source code would vary depending on the programming language that was used. Learning a programming language may seem difficult at first, but it will become easier with practice.

STEP 5: Test the Program:

Once you have a program written that compiles, you *need to make sure that it solves the problem that it was intended to solve and that the solutions are correct*. Running a program is the process of telling the computer to evaluate the compiled instructions.

When you run your program, if all is well, you should see the correct output. It is possible, however, that your program works correctly for some set of data input but not for all. If the output of your program is incorrect, it is possible that you did not convert your algorithm properly into a proper program. It is also possible that you did not produce a proper algorithm back in step 3 that handles all situations that could arise. Maybe you performed some instructions out of sequence. Whatever happened, such problems with your program are known as bugs. *Bugs are problems/errors with a program that causes it to stop working or produce incorrect or undesirable results*.

You should fix as many bugs in your program as you can find. To find bugs effectively, you should **test** your program with many test cases (called a test suite). It is also a *good idea to have others test your program* because they may think up situations or input data that you may never have thought of. The process of finding and fixing errors in your code is called **debugging** and it is often a very time-consuming "chore" when it comes to being a programmer. If you take your time to carefully follow problem solving steps 1 through 3, this should greatly reduce the amount of bugs in your programs and it should make debugging much easier.

STEP 6: Evaluate the Solution:

Once your program produces a result that seems correct, you need to re-consider the original problem and make sure that the answer is formatted into a proper solution to the problem. It is often the case that you realize that your program solution does not solve the problem the way that you wanted it to. You may *realize that more steps* are involved. Effective programs don't happen by accident — they are the result of keen observation and forethought.

<Best Regards>

Dr. Raaid Alabady