

Programme Development: Problem Analysis

Introduction:

A good program is not just about the code. Understanding the problem you're trying to solve and identifying the program's requirements is essential. So far, we have looked at some of the programming structures that make code possible. This module focuses on the more conceptual aspects of designing complex programs, you will learn how to combine a programming language with the steps that every good programmer takes when starting a new project. You will also learn how to represent the logic of a program in a graphical manner using a flowchart. A flowchart is a tool that assists with solving a programming problem before you attempt the program. All this will help you to become a professional programmer.

Problem Description:

Have you ever had a conversation in which you said, "Oh! That's easy! If only you had told me what you wanted!" Just about everyone has.

When you start writing a computer program, finding out exactly what needs to be done is very important. In this section you are going to learn some of the things you must consider before you start to write a new program.

Before you start, you will probably have some idea of what the user wants it to do — we call this a user requirement. Some examples of user requirements are:

- a program that calculates how much money you have saved.
- a program that keeps track of what actors are in your favourite films, and
- a program that stores customer records at the hair salon and sends them email reminders when it's time to have their hair cut.

As you can see from the list, some user requirements are useful, some are fun, and some are ridiculous. When you start thinking about your program, you will quickly realise that you need a lot more information in order to get started. For example, how will you input the information? Does it need to be stored? How fast does hair grow?! You need to find out more of the system requirements,

The first step in any programming task is to clearly understand the problem. This process involves asking detailed questions to ensure that the user's requirements are fully understood. Considerations include:

- **User requirements:** Identify what the user wants the program to do (e.g., track savings, store customer records).

- **System requirements:** Clarify inputs, outputs, processing, and storage needed for the program.

The IPO Model:

You should think about all your computer programs in terms of input, storage, processing and output. This is known as the Input-process-Output (IPO) model. In most cases, a computer takes some sort of data input — for example, your name and date of birth entered on the keyboard. It will then store the data, usually by saving it to the hard drive to use later. It will process the data - for example, compute if today is your birthday. Finally, the program will give some output, such as displaying 'Happy Birthday Michael!' on the screen,

The **Input-Process-Output (IPO)** model helps clarify the structure of your program. It focuses on the flow of information:

- **Input:** Where does the data come from (e.g., user input or files)?
- **Process:** What does the program do with the input (e.g., calculations)?
- **Output:** What result does the program produce (e.g., display on the screen, save to a file)?

Using the IPO model early in development will guide your system requirements and help you plan the overall logic.

Program Requirements:

Identify the program requirements.

For every program that you write, you should think about input, storage, processing and output and find out how these affect your program. This will allow you to develop system requirements, such as:

- Input requirements: Where will the data come from? Does someone type it in? Who? Is it from a computer file? What is the name of the file? Where is the file? Do you get it on the Internet?
- Storage requirements: Does the data need to be stored? Is there a lot of data? For how long must it be stored? Are there different types of data? Is the data very important? Is the data private?
- Processing requirements: What does the program need to do? Does it need to do it many times? Do you understand all the calculations it needs to do? What happens if the calculations go wrong?
- Output requirements: Who will use the program output? Should it be saved to a file, displayed on the screen or sent in email? Should it be printed as a hard copy in colour, or in black and white?

You need to make sure at this stage that you find out exactly what needs to be done. To do this, you need to ask good, clear questions. This is sometimes difficult because the user might not fully understand the problem, or hasn't thought it through completely. Perhaps the user who gave you the requirement is not the only user, or there may be other users who have different ideas. So you need to ask many questions about what the input, storage, processing and output requirements are.

Here is an example of a user requirement given to you by a teacher:

I have a big file with all the class marks, and I want a computer program to take the marks and rank the students.

Let's think about the four system requirements — input, storage, processing and output — and then think of some questions that you might ask the teacher to get more information:

- Input requirements: Where is the file located? What is its name? What is the format of the file?
- Storage requirements: How many marks are there? Do you want to keep these records forever? Will you need to add more results later? Will you need to remove some marks? Do the marks have to be kept secret?
- Processing requirements: Do you want to rank the marks highest to lowest? What if two people have the same mark? Is there anything else you want - perhaps the average mark?
- Output requirements: Do you want the output printed to the screen, saved in a separate file?

Imagine that you have asked the teacher these questions. She says that she will give you the file, but your program should ask for the file name when it starts. The file contains the student names, the date they wrote the test and their mark.

Before coding, define the following:

1. **Input requirements:** Determine the source of data.
2. **Storage requirements:** Decide if and how data should be stored.
3. **Processing requirements:** Understand the calculations or actions to be performed.
4. **Output requirements:** Identify what the program will deliver and to whom.

Example Scenario:

Draw an IPO chart that represents the solution to the following problem:

A builder requires a program that will help determine the number of boxes of tiles required to tile the floor of a room. In order to do this, the program would need to calculate the area of the floor. The number of tiles required would need to be calculated based on the size of the tile. You can only purchase full boxes of tiles and not parts Of a box.

Sample

Completed IPO Chart

Figure 2-10: Completed IPO chart

Input	Processing	Output
current weekly pay raise rate	Processing items: weekly raise Algorithm: 1. enter the current weekly pay and raise rate 2. calculate the weekly raise by multiplying the current weekly pay by the raise rate 3. calculate the new weekly pay by adding the weekly raise to the current weekly pay 4. display the new weekly pay	new weekly pay

An Introduction to Programming with C++

17

Input:

100.0

0.01

Output:

101.00

Code

Module Program

Sub Main()

Dim weekly_pay As Double

Dim raise As Double

Dim weekly_raise As Double

Dim new_weekly_pay As Double

' 1. current weekly pay:

Console.WriteLine("Enter weekly pay: ")

weekly_pay =

Convert.ToDouble(Console.ReadLine())

' 2. raise rate:

Console.WriteLine("Enter raise rate (e.g.,

0.1 for 10%): ")

raise =

Convert.ToDouble(Console.ReadLine())

' 3. weekly raise:

weekly_raise = weekly_pay * raise

' 4. new weekly pay:

new_weekly_pay = weekly_pay +

weekly_raise

' 5. Output:

Console.WriteLine("New weekly pay

is: {0:N2}", new_weekly_pay)

' Pause to view the result

Console.WriteLine("Press any key to

exit...")

Console.ReadKey()

End Sub

End Module

Consider a teacher who wants a program to rank student marks. **Questions to ask:**

- **Input:** Where is the file with the marks? What format is it in?
- **Storage:** How many records? Should they be stored indefinitely?
- **Processing:** How should marks be ranked? How to handle ties?
- **Output:** Should the results be printed or saved?

In this case, after clarifying these details, a program can be developed to process and display student rankings.

Using an IPO Chart:

An **IPO chart** summarizes the steps needed to solve a problem. For example, to calculate the area of a rectangular field:

- **Input:** Length and width of the field.
- **Process:** Multiply length by width.
- **Output:** Display the area in square meters.

Example Problem - Waiter's Pay Calculation:

A restaurant owner wants a program to calculate each waiter's pay based on hours worked, hourly rate, and tips.

- **Input:** Hours worked, hourly rate, total tips.
- **Process:** Multiply hours worked by hourly rate, divide tips equally, and calculate total pay.
- **Output:** A payslip for each waiter.

<ul style="list-style-type: none"> • Input: The user enters: <ul style="list-style-type: none"> ○ <code>hoursWorked</code>: Total hours the waiter worked. ○ <code>hourlyRate</code>: The pay per hour for the waiter. ○ <code>totalTips</code>: The total amount of tips received by all waiters. ○ <code>numberOfWaiters</code>: The number of waiters working to calculate the share of tips. • Process: <ul style="list-style-type: none"> ○ <code>wages</code>: Calculate the wages by multiplying the <code>hoursWorked</code> by the <code>hourlyRate</code>. ○ <code>tipsPerWaiter</code>: Divide the <code>totalTips</code> by the <code>numberOfWaiters</code> to get tips per waiter. ○ <code>totalPay</code>: Add <code>wages</code> and <code>tipsPerWaiter</code> to get the waiter's total pay. • Output: <ul style="list-style-type: none"> ○ The program prints a payslip that includes the hours worked, hourly rate, wages, tips, and total pay. 	<pre> Module WaiterPayCalculator Sub Main() ' Declare variables Dim hoursWorked As Double Dim hourlyRate As Double Dim totalTips As Double Dim numberOfWaiters As Integer Dim wages As Double Dim tipsPerWaiter As Double Dim totalPay As Double ' Input: Get hours worked, hourly rate, total tips, and number of waiters Console.WriteLine("Enter the number of hours worked: ") hoursWorked = Convert.ToDouble(Console.ReadLine()) Console.WriteLine("Enter the hourly rate: ") hourlyRate = Convert.ToDouble(Console.ReadLine()) Console.WriteLine("Enter the total tips: ") totalTips = Convert.ToDouble(Console.ReadLine()) Console.WriteLine("Enter the number of waiters: ") numberOfWaiters = Convert.ToInt32(Console.ReadLine()) ' Process: Calculate wages and tips per waiter wages = hoursWorked * hourlyRate tipsPerWaiter = totalTips / numberOfWaiters totalPay = wages + tipsPerWaiter </pre>
--	---

	<pre> ' Output: Display the payslip for the waiter Console.WriteLine(vbCrLf & "Waiter's Payslip") Console.WriteLine("-----") Console.WriteLine("Hours Worked: {0}", hoursWorked) Console.WriteLine("Hourly Rate: {0:C2}", hourlyRate) Console.WriteLine("Wages: {0:C2}", wages) Console.WriteLine("Tips per Waiter: {0:C2}", tipsPerWaiter) Console.WriteLine("Total Pay: {0:C2}", totalPay) ' Pause to view the result Console.WriteLine(vbCrLf & "Press any key to exit...") Console.ReadKey() End Sub End Module </pre>
Output	<pre> Waiter's Payslip ----- Hours Worked: 40 Hourly Rate: R15.00 Wages: R600.00 Tips per Waiter: R50.00 Total Pay: R650.00 </pre>

Problem-Solving with Flowcharts:

Flowcharts visually represent the flow of a program. By using standard shapes, such as rectangles for processes and diamonds for decisions, flowcharts allow you to map out the logic before coding. For example:

- **Rectangles** represent actions (e.g., calculate wages).
- **Diamonds** represent decisions (e.g., does the person qualify for a discount?).



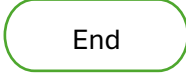



Example Flowchart:

For a program that determines if a customer qualifies for a pensioner discount:

- **Input:** Age of the customer.
- **Decision:** If age is 65 or above, apply a 10% discount.
- **Output:** Display whether the customer qualifies for the discount.
-
-

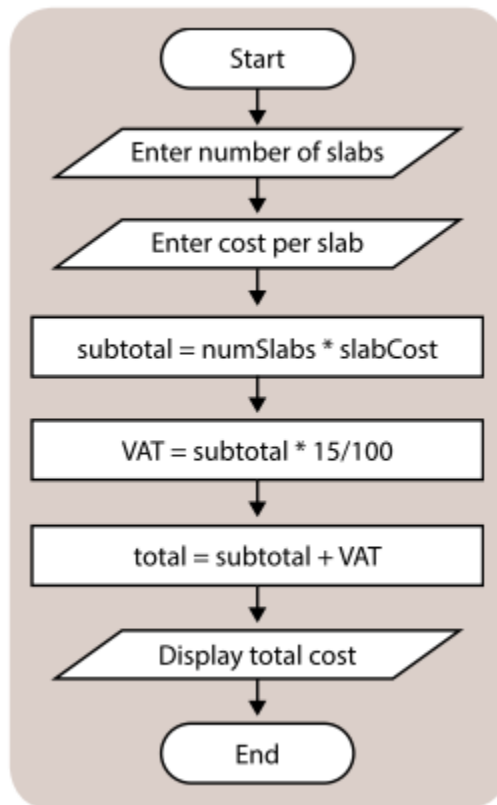
Represent a solution to a given problem in flowchart flowchart

- Flowcharts provide a visual means to show the flow and logic of a computer program. A flowchart is a visual tool that you solve a programming problem. It saves time during the coding process - because the problem has already been solved, the simply needs to translate the solution into code, algorithm is represented using a series of shapes that have certain meanings.
- same conventions are used throughout the world, so flowcharts present a universal way of representing an algorithm. A flowchart is easier to understand than trying to read actual programming code. Flowcharts are often used to represent small sections of code within a larger program. The focus is then on the sections that are the most complex or involve the most logic.
- Flowchart symbols**
 - There are many flowchart symbols to represent different types of actions. This table shows the most used symbols.

Linking lines with an arrow at the one end are used to show the direction of flow of the program and to link the various symbols to each other.	
Terminator blocks are used to show the start and end Of a section In a program.	 
Rectangles are used for actions, instructions or processing.	
Parallelograms are used to show input and output.	
Diamond shapes are used to show decision-making.	

Flowchart examples

An example of a flowchart for a program in which a stallholder at a farmers' market enters the number of slabs of handmade chocolate a client wants to buy as well as the price per slab. The program displays the total cost of the chocolate, including VAT at a rate of 15%.



Pseudocode:

Pseudocode is used to represent the solution to a problem in a series of steps using ordinary language. The algorithm is expressed using simple, clear steps. Note that the use of programming code is not permitted in pseudocode. "The idea is that a programmer should be able to code a solution using any programming language based on the Steps set out in the pseudocode. Each Step needs to be a single, simple instruction.

The pseudocode will make use Of the same structures for making decisions and repetition as those used in the flowcharts in the previous section. We are going to look at the same examples to see how the flowcharts be represented using pseudocode.

Pseudocode is a simple, non-programming language used to outline the steps of a program. For example, calculating the total cost of an item with VAT might look like this:

BEGIN

1. numSlabs \leftarrow number of slabs entered

2. slabCost \leftarrow cost per slab entered

3. subtotal \leftarrow numSlabs * slabCost

4. VAT \leftarrow subtotal * 0.15

5. total \leftarrow subtotal + VAT

6. Display total

END

Decision Structures in Pseudocode:

Use **if statements** in pseudocode for decision-making. For instance, offering a 5% discount if a customer buys 10 or more slabs:

```
IF numSlabs >= 10
```

```
    subtotal ← subtotal * 0.95
```

```
END IF
```


Repetition and Loops:

For repetitive tasks, use loops such as `while` or `for`. For example, a `while` loop could keep adding numbers entered by the user until they type “no”:

```
WHILE answer = "yes"
```

```
    Add number to total
```

```
END WHILE
```

Conclusion:

Effective problem analysis and use of tools like IPO charts, flowcharts, and pseudocode allow you to systematically solve problems and design programs. Each step from understanding user requirements to coding can be greatly enhanced with these methods, ensuring your program meets all its objectives.