Program Design

Bill Newman - updated 01/07/2016

Design Steps: Programs have been designed for so long that general steps can be assigned to the creation of a program:

1. **Requirements Analysis**: Identify business needs, inputs, outputs, processes
2. **Outline**: Define and visualize the program
3. **Algorithm**: Develop the outline into an algorithm (A set of steps to accomplish the task)
4. **Desk Check**: Pre-test any questionable algorithm steps
5. **Code**: Write, document and test the program
6. **Test**: Run and test the code under all possible circumstances
7. **Maintain**: Maintain and refactor (improve) the program

Below are some notes on these steps:

Requirements Analysis -- Understanding precisely what the customer needs involves time and effort. Often the customer does not know exactly what is needed, or how to explain themselves clearly. Work to obtain all the information so you solve the **right** problem. Frequently **it pays to go back to the customer several times** to get the needed information. Developing a questioning technique for the customer is very important. Here are some sample questions:

- What do you want the program to do? (output, process)
- What will the person who uses it type in? (input)
- How exact do you need this information to be? (precision, size)
- How often will you use this program? (flexibility, cost viability)
- Do you see yourselves using this program for anything else? (scope of program)
- How long do you see yourselves using this program? (language used, long term viability)

When a customer brings up nouns repeatedly (customers, orders, books, etc.) each of these can become a potential entity, which could develop into a database table and later into a class, in OOP terms.

When programs become more complex, it can help to write user stories that identify the roles and requirements clearly.

Outline -- Once you have identified the needs of the customer, create a design document to describe the program specifications. The document should give a fairly complete description while maintaining a high-level view of the project.

Identify the goals and challenges of the program. For problematic aspects, consider creating an **Input Process Output chart** (IPO, see below).

Input Process Output Chart

Input	Process	Output
degrees F	1. Get input from user via command line 2. Calculate using the formula C = 5/9 * (F-32) 3. Output data to user via command line	degrees C

This provides a visualization of the problem. Break a large program down into smaller tasks, which could become functions or separate classes. Each separate section could have its own IPO.

Algorithm -- Write the step by step process to be followed by the program. Here are some simple example steps:

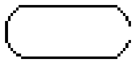
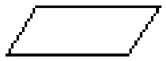
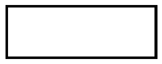
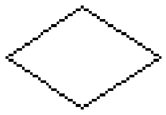
1. Ask for input from the user
2. Check the user input for valid data
3. Process the data for the user
4. Provide output to the user

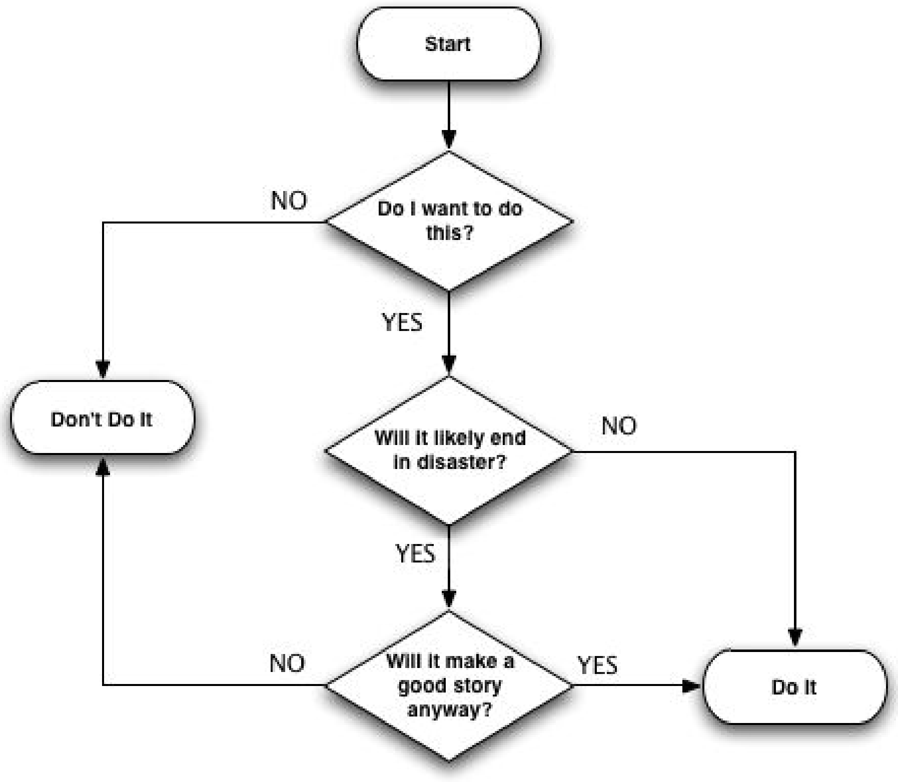
Each step could then be broken down to further sub-steps. Eventually the steps can be so precise that it almost looks like code. At this point the broken down steps can be called pseudocode.

Well written pseudocode can be copied later into our program and then each line could be written accomplish the pseudocode steps.

Flowcharts: When we first design a program, we imagine everything to go as planned. However, we need to accommodate user errors, or multiple logical paths. In order to show how your program will accommodate such needs, consider creating a flowchart.

Below are some of the symbols you can use to indicate flow and structure:

-  oval -- start/stop symbol
-  parallelogram -- input/output symbol
-  rectangle -- processing symbol
-  diamond -- decision symbol


Start
Do I want to do this?
NO
YES
Don't Do It
Will it likely end in disaster?
NO
YES
Will it make a good story anyway?
NO
YES
Do It

You may also wish to create a UML Use Case Diagram.

If we're thinking in terms of a website, we should also consider Page Flows and Funnels

Desk Check -- The person performing the desk check effectively acts as the computer, using pen and paper to record results. The desk checker carefully follows the algorithm to adhere to the program's logic. The desk check can expose problems with the algorithm.

When you carefully analyze the problem, then write a comprehensive algorithm, and finally desk-check it with a variety of data, you will find most, if not all, of the logic errors. With these out of the way, your coding and testing will go much more quickly and smoothly. Be sure to choose your data for desk-checking carefully to obtain the expected answer to each set of data before running the data through the Algorithm. Ask yourself questions like these:

- What if the user inputs unexpected data?
- How will I provide feedback to the user?
- Will the data types properly store the data?
- Are the numeric data types of the proper precision?
- Will the program always respond with proper english? (plurals, etc.)
- How will I test my data?

The desk check can also utilize users in paper prototyping to look over the logic of your program before it's built and identify issues and preferences.

Coding: Now it is time to actually begin coding your algorithm into a program. In general, each statement in the algorithm/pseudocode requires one or more lines of programming code. You will frequently run and test your program at intervals. **The more often you test small pieces of your code, the less troubleshooting you will need to do.** Breaking your program down to testable pieces is called unit testing.

It is usually best to consider documenting your application in a formal manner as you build your application. This is because you'll never understand it better than when you're building it: A beginner's guide to documentation

Software Testing -- While you code you will run tests on your program, but at the end others may run the program through a rigorous series of software tests. These tests are designed to give one last pass to your program to be sure it is safe and secure for the public. Software testers sometimes use special software designed to expose the flaws in your program. If problems occur in your program during this phase, you will be coding and testing until all is satisfactory.

Maintenance -- Once your program is complete, you will likely create documentation for the program. The IPO charts with either pseudocode, flowcharts, or both are important documentation for your program. Documentation created while the program is built ensures all needs are met, and all important areas of the program are noted. You may also be asked to write instructions on how to use your program.

If your program is designed for a long shelf life, you may be asked to refactor your program, where you to re-examine the program to improve, expand or upgrade it.

Resources

Fundamentals of Program Design - This is a copy of a lecture from David Schmidt of Kansas State University on program design

Software Design - Wikipedia on step 2 of the software process. Great detail!