PROCEDURAL DESIGN METHODOLOGY

A ***design methodology*** combines a systematic set of rules for creating a program design with diagramming tools needed to represent it. ***Procedural design*** is best used to model programs that have an obvious flow of data from input to output. It represents the architecture of a program as a set of interacting processes that pass data from one to another.

Design Tools

The two major diagramming tools used in procedural design are data flow diagrams and structure charts.

Data Flow Diagrams

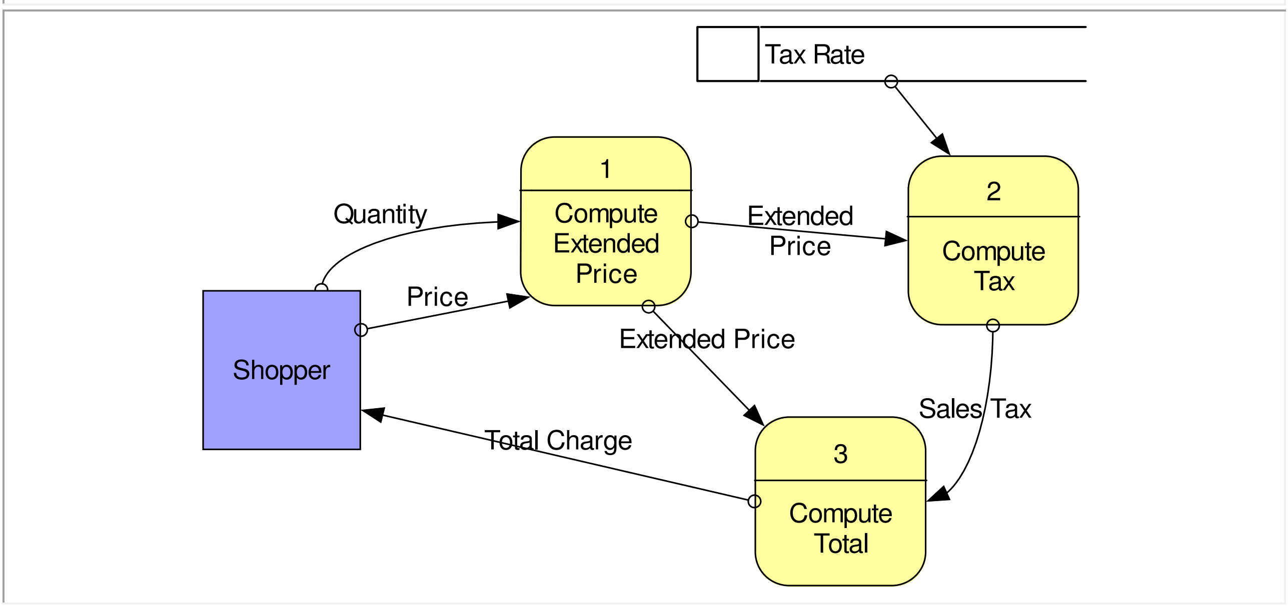
A ***data flow diagram*** (or ***DFD***) is a tool to help you discover and document the program's major processes. The following table shows the symbols used and what each represents.

Data Flow Diagram Symbols (showing the two major symbol sets)			
Name	**Gane and Sarson Symbol**	**Yourdon Symbol**	**Description**
Process	# / Process Goal	# Process Goal	A major task that the program must perform
Data Flow	Name	Name	Data that flows into and out of each process
Data Store	ID Description	Name	An internal data structure that holds data during processing
External Entity	Name	Name	Devices or humans which input data and to which data is output

The DFD is a conceptual model – it doesn't represent the computer program, it represents what the program must accomplish. By showing the input and output of each major task, it shows how data must move through and be transformed by the program.

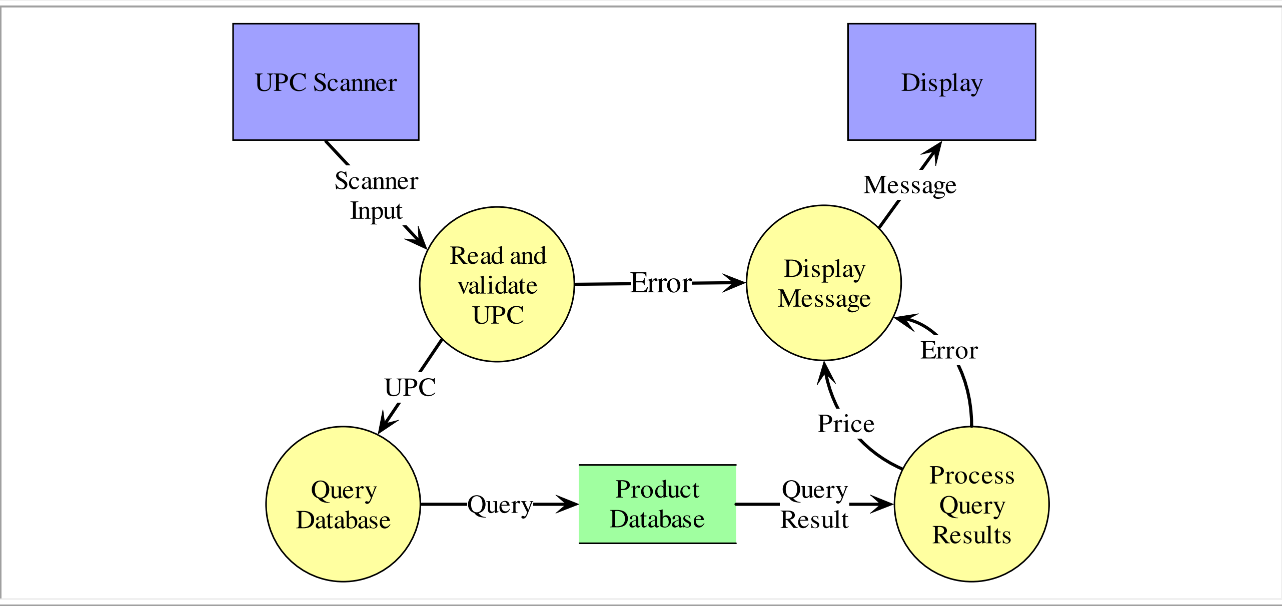
Example

This DFD uses Gane and Sarson symbols to show what's involved in calculating a shopper's total charge given a quantity and price. For example, 2 candy bars @ 79¢ apiece with 6% sales tax tallies to $1.67. The numbers on the processes are for identification only; they do not indicate order of execution.



Example

This DFD uses Yourdon symbols to model a department store's price checking machine.



Structure Charts

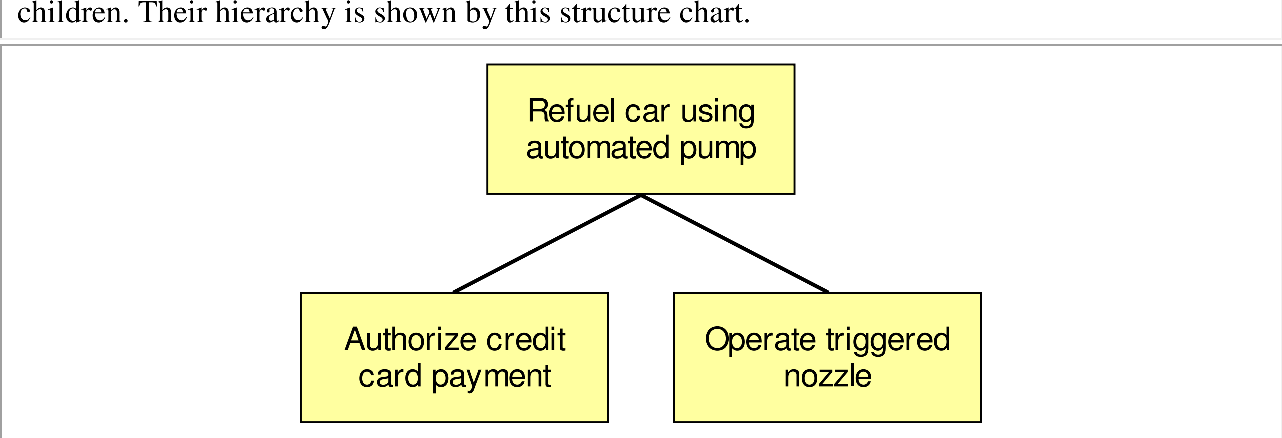
A ***structure chart*** is a tool to help you derive and document the program's architecture. It is similar to an organization chart.

Structure Chart Symbols	
Symbol	Description
Component	A major component within the program
	Connects a parent component to one of its children
Name	Data that is passed between components

When a component is divided into separate pieces, it is called the ***parent*** and its pieces are called its ***children***. The structure chart shows the hierarchy between a parent and its children.

Example

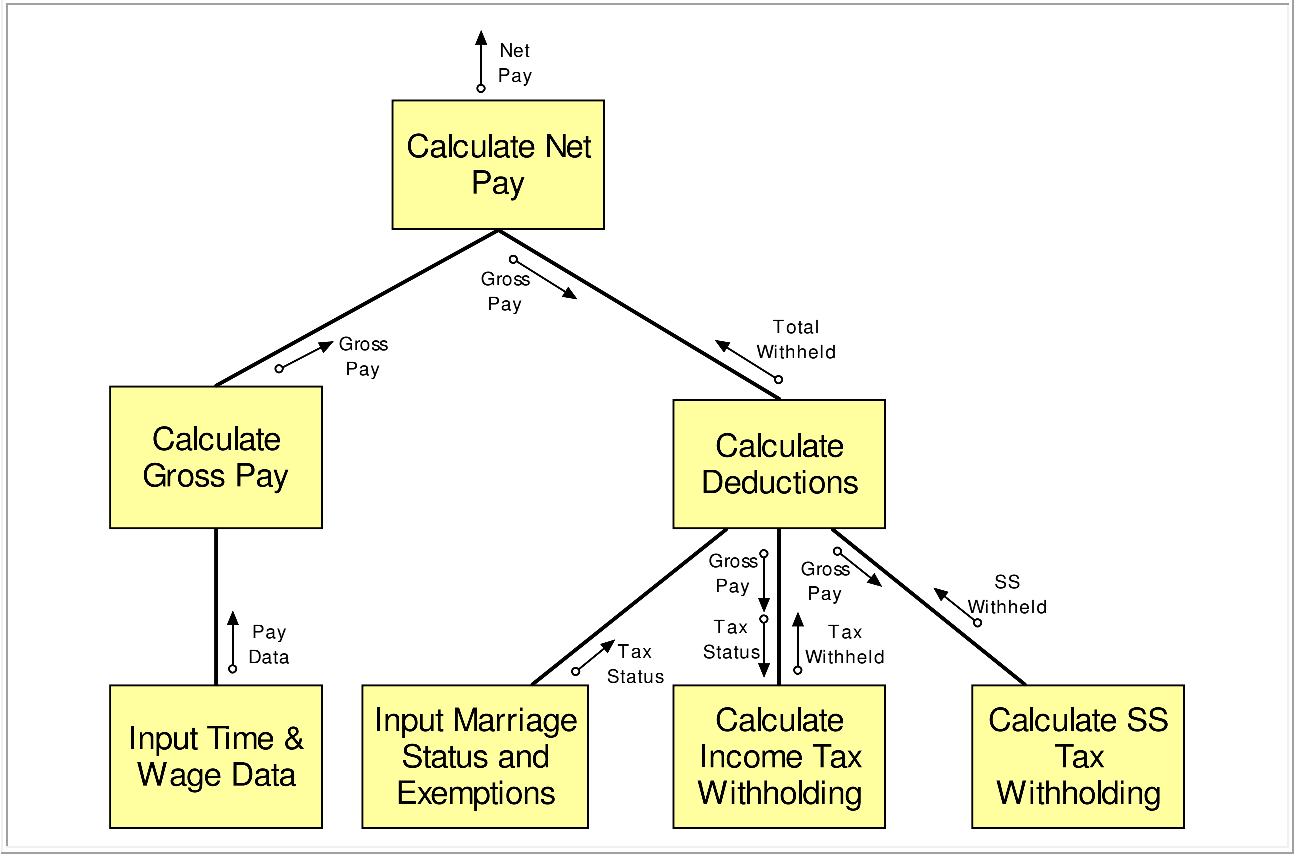
To refuel your car using an automated pump you must (1) authorize the credit card payment and (2) operate the pump trigger to fill your tank. *Refuel car* is the parent task; the other two are its children. Their hierarchy is shown by this structure chart.



As shown in the above example, a structure chart can be used to show the relationship between conceptual tasks. On the other hand, it can also be used to show the organization of a computer program into its physical components.

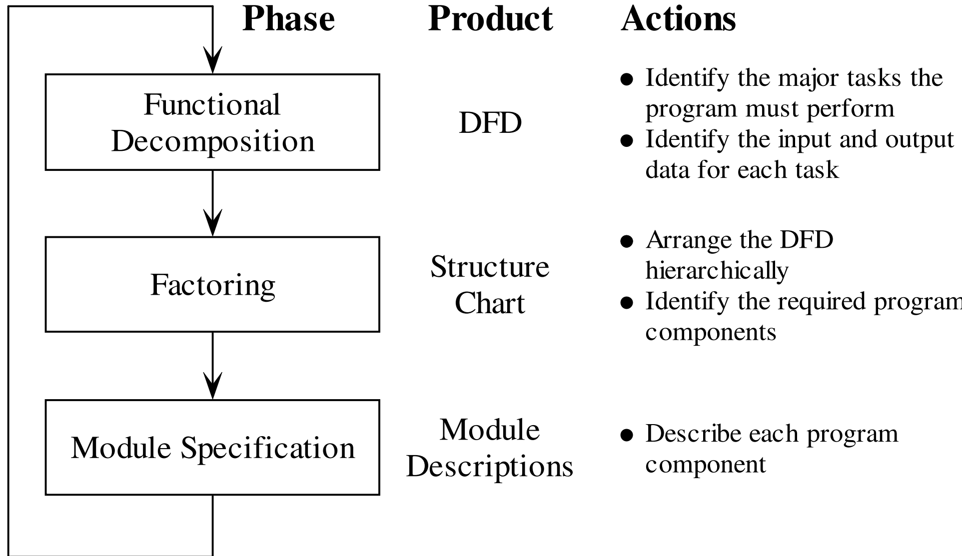
Example

This structure chart shows the hierarchical relationship between the methods within a computer program. Here the line symbol represents a method call and the data arrows represent arguments and return values. For instance, it shows that the program has a method to *Calculate Deductions* that receives the *Gross Pay* as an argument and returns the *Total Withheld*.



Design Methodology

Here are the basic steps you follow to create a procedural design.

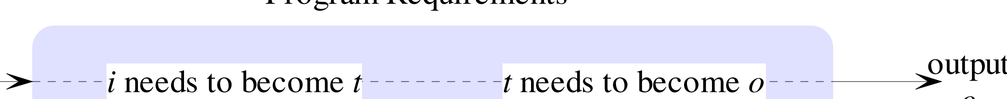


Functional Decomposition

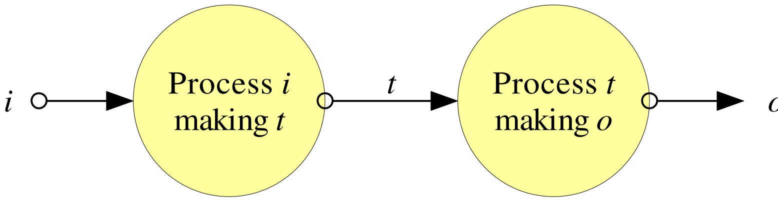
In computer programming, ***decomposition*** is the process of dividing a large entity into more manageable pieces. For a procedural design, this means dividing tasks into sequences of smaller tasks, which is ***functional decomposition***.

One technique for doing this, called ***data flow analysis***, involves (1) identifying a major data flow, (2) following it from input to output, (3) determining where it undergoes a major transformation and (4) dividing the processing at that point. To illustrate, given the following program requirements:

Program Requirements



Data flow analysis yields:



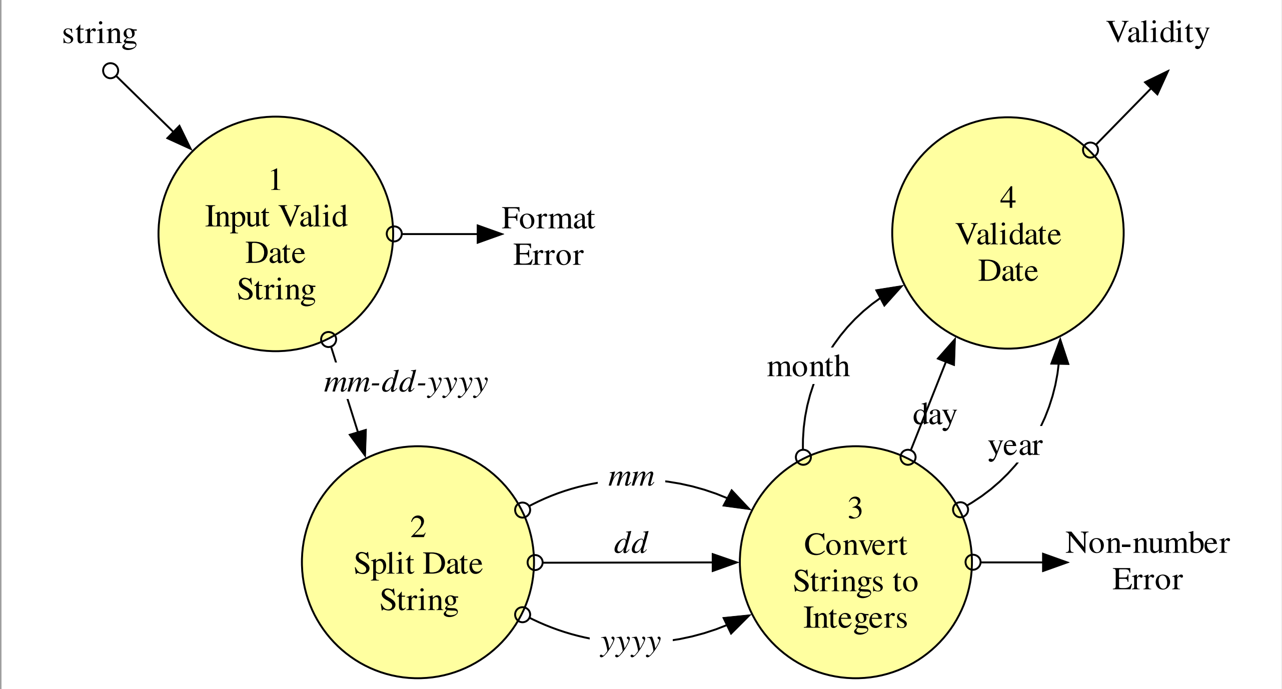
Example

Suppose our program is required to input and validate a calendar date. The input is a string in the form *mm-dd-yyyy* or *mm/dd/yyyy*. For example, *12-25-2012* is a valid date, *02-29-2013* is not because no such date exists and *2-8-13* is not because it isn't in the correct form.

Assuming that the input string is in the required form, there are two major transformations in the date: (1) the input string must be separated into its *mm*, *dd* and *yyyy* segments and (2) each segment must be converted to an integer:

```
                   "12"          12
"12-25-2012"  -->  "25"   -->    25
                  "2012"        2012
```

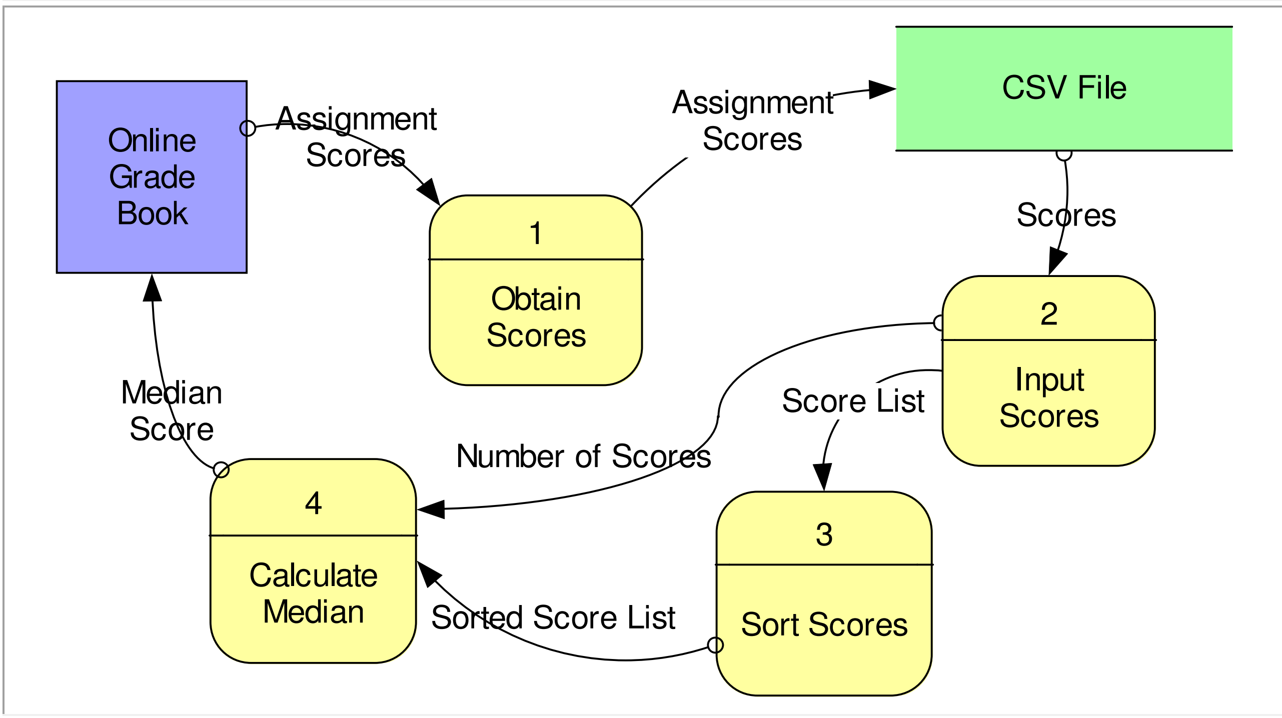
Once the month, day and year is obtained, the date can be validated. The required processing is shown in the following DFD.



Example

Suppose our program is required to obtain a list of assignment scores from a college's online grade book and calculate the median score. The median is the middle; there must be the same number of scores above the median as below it. This requires that the scores be sorted into numerical order. If there is an odd number of scores then the median is the score in the middle. For example, 73 is the median of 55, 65, 73, 85 and 97. If there is an even number of scores then the median is the average of the two in the middle. For example, 75 is the median of 55, 65, 85 and 97.

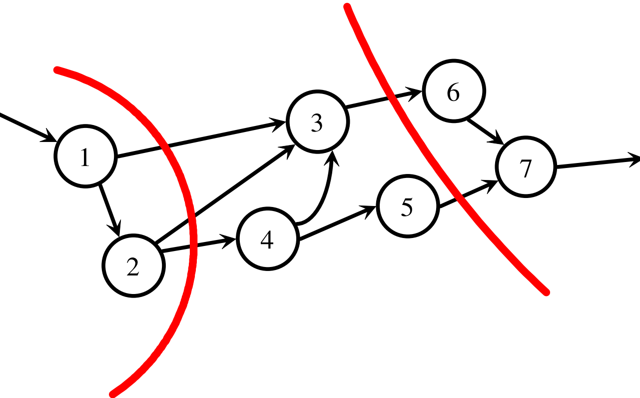
The DFD below shows the major transformations that must take place. (1) The scores are obtained from the online grade book and placed into a CSV file stored on the external drive. From there, (2) the scores can be input, (3) sorted and (4) the median score calculated.



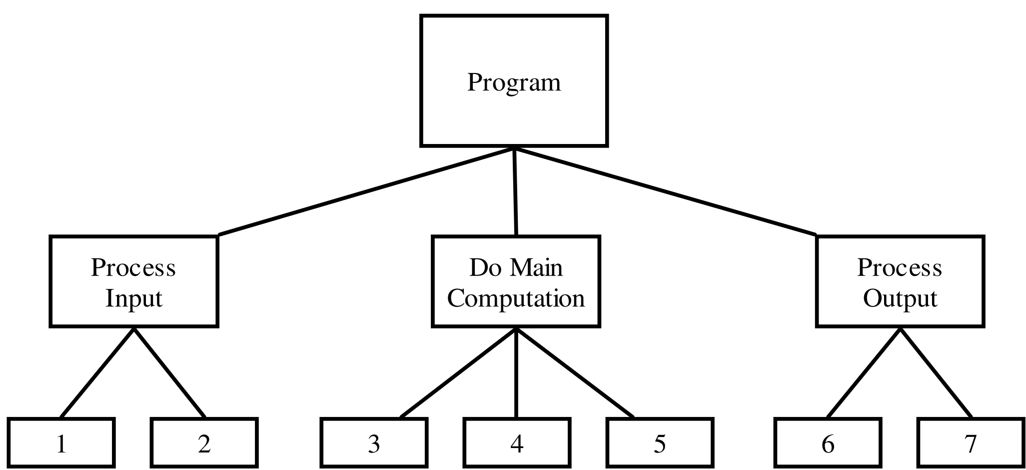
Factoring

Factoring is the second phase of procedural design in which you create a structure chart that shows what program components need to be implemented. You do this in two passes. First, arrange your DFD hierarchically. Second, identify exactly which conceptual processes are to be implemented as physical components in the program.

To arrange your DFD hierarchically, cut it into three partitions: (a) processes that prepare input for the main computation, (b) processes that perform the main computation and (c) processes that prepare the output.



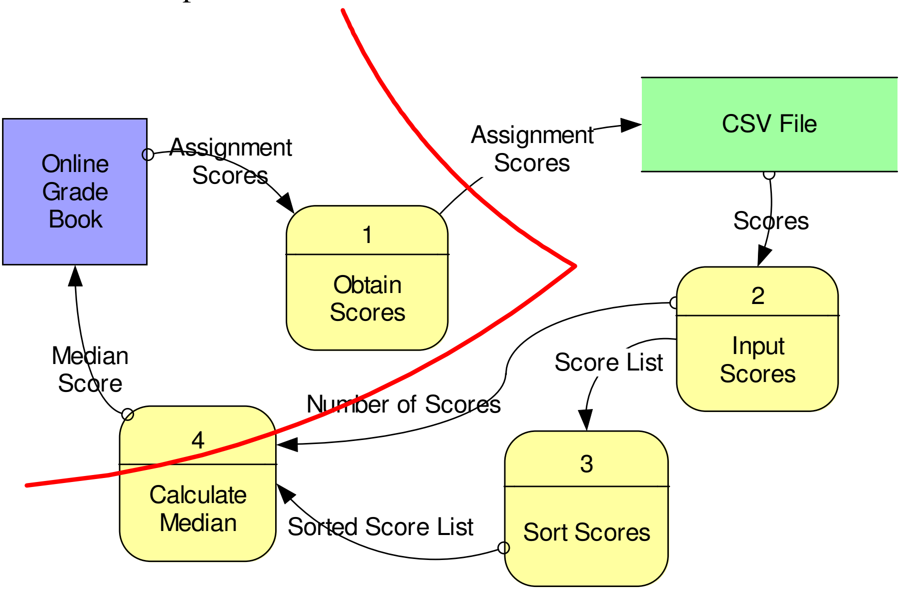
Organize the processes under one or more “managing” processes that control the flow of the computation. The above DFD becomes:



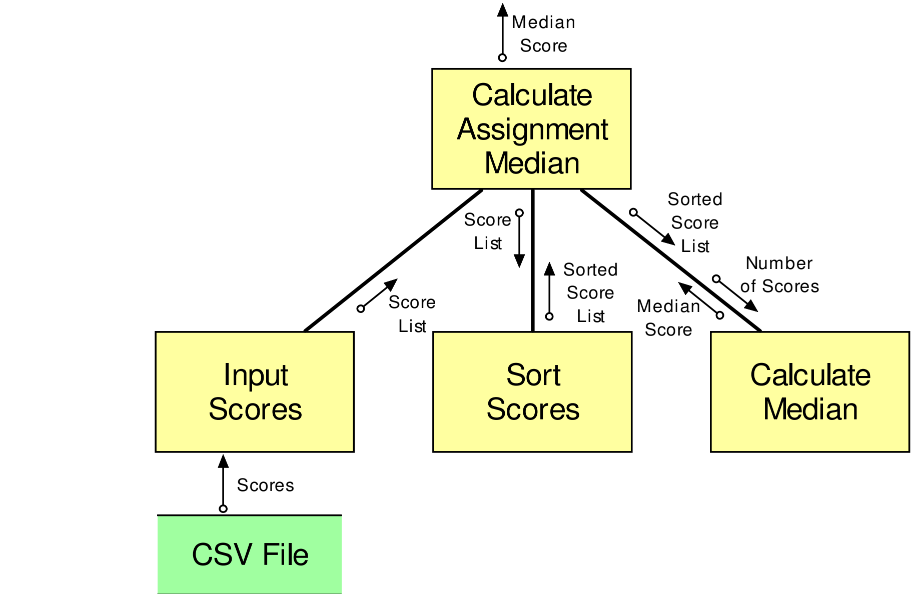
To identify the physical components you want to implement, take a good close look at (1) the overall organization of the structure chart and (2) each process in it. Be prepared to (a) explode a process into several methods, (b) combine several processes into a single method or (c) explode and recombine parts of different processes.

Example

Let's factor the median finding program. Following the principle of information hiding, we must split parts of the program that must interface with the online grade book into a separate module. This includes the *Obtain Scores* process and the part of the *Calculate Median* process that gives the result back. Since this module requires knowing how to interface with the grade book, we'll not deal with it in this example.



What remains are processes 2, 3 and the median calculating part of process 4. Each of these readily maps into an input, transform and output method within a structure chart. I've added a manager method above them to direct the sequence of computations.

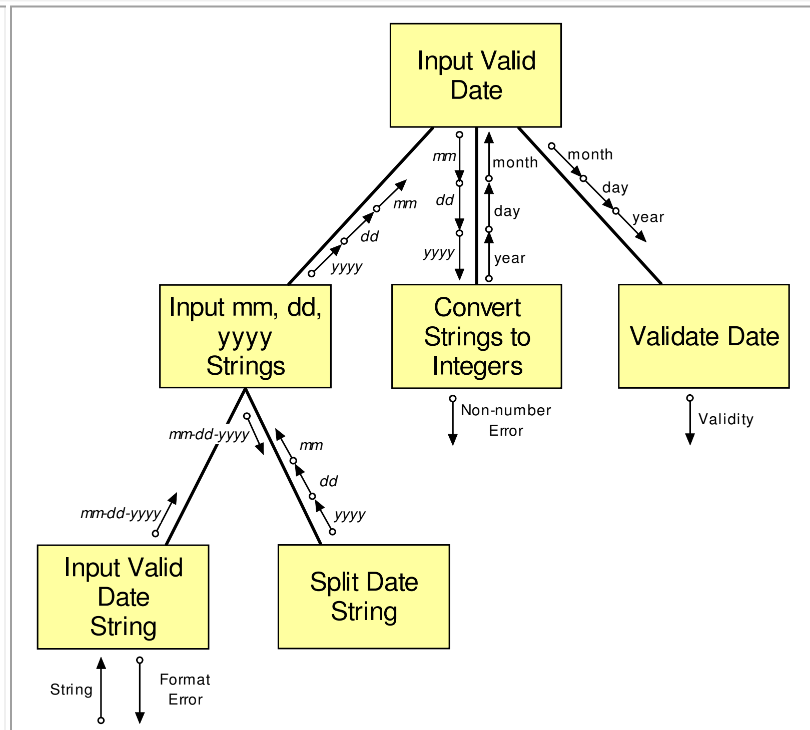


Example

Let's factor the date validation program.

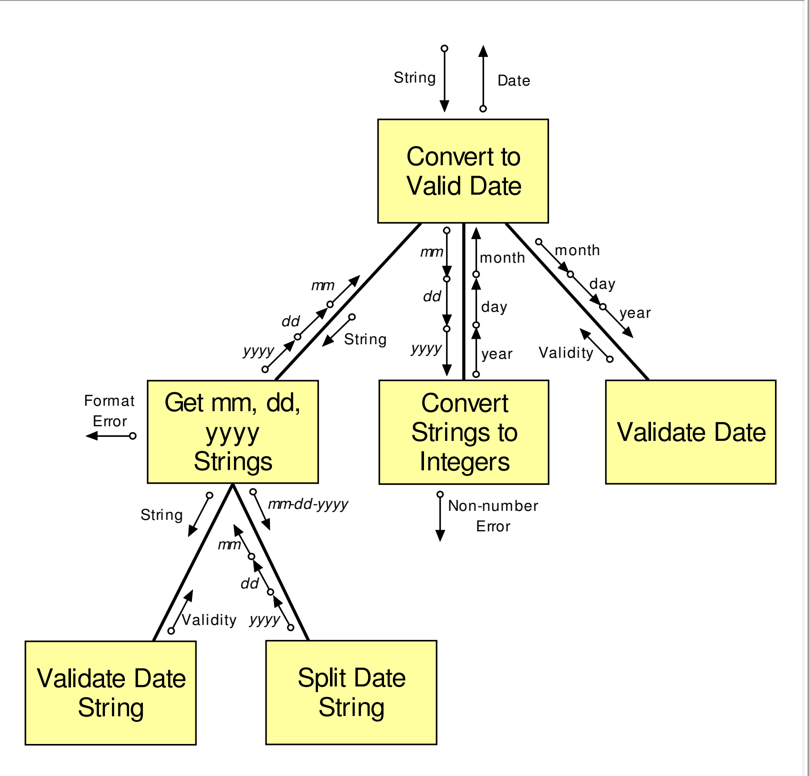
Step 1 is to arrange the DFD hierarchically. I decide that processes 1 and 2 are "input," process 3 is "computation" and process 4 is "output." It doesn't matter whether or not I get this choice "right" because soon I'll work to adjust the hierarchy. I arrange the processes under appropriate "managing" processes. Finally, I add the data flows using the DFD as my guide.

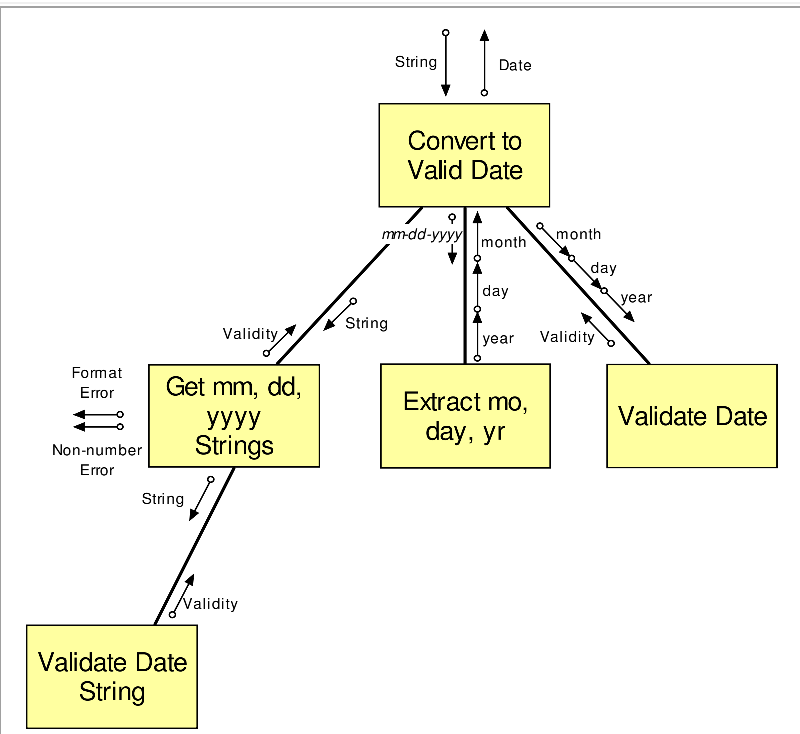
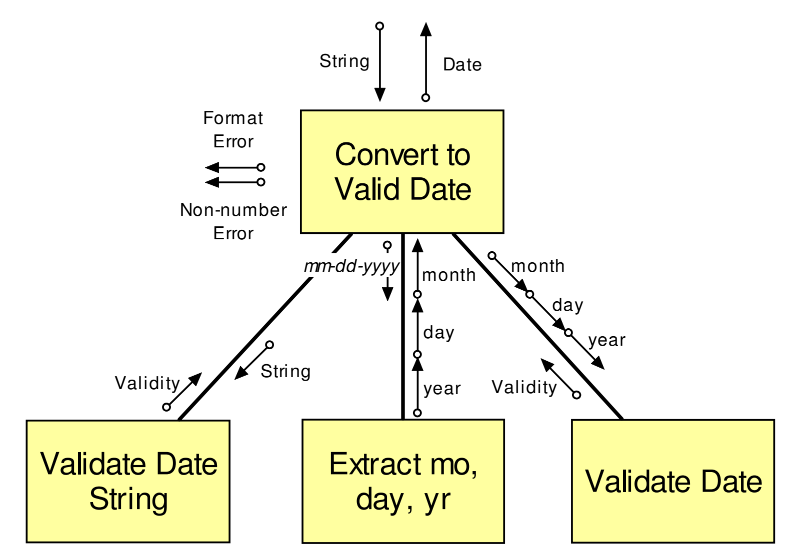
The resulting diagram is at right.



Step 2 is to identify the actual program components that I want to implement. The first alteration is to make the entire unit one that accepts the string as an argument and returns the valid date instead of one that does input and output. Such a component could be used in a number of contexts – for strings that are input or come from a GUI or whatever. The errors can be signaled using exceptions.

The revised diagram shows the data as parameters rather than input and output values.



Next, I notice the excessive number of couplings involving *String* and its substrings *mm*, *dd* and *yyyy*. I cut these down by combining *Split Data String* and *Convert Strings to Integers* into one component. This makes sense if a validation method has already determined that the string is in the form *mm-dd-yyyy*.	
Now there's too little fan-out on the left side of the diagram. In addition, there seems little need for the *Get mm, dd, yyyy Strings* component. So I combine it with its child. This diagram is my final structure chart.	

Module Specification

Module Specification is the act of documenting your program design by fully describing each of its modules. ***Module*** is a general term that can refer to any manner of computer program components, including a single method, a single class, a single object or a collection of related methods.

When the module is a single method, the following facts must be provided. Seasoned programmers generally write them using a combination of computer language and English.

- The method's name
- A description of what it does
- Its number of arguments and the data type and purpose of each
- A description of the return value and its data type
- A description of any exceptions that it may throw.

It is often useful to specify the behavior of a method by stating its preconditions and postconditions. A ***precondition*** is a statement of what must be true when the method is called; a ***postcondition*** is a statement of what must be true when the method returns.

Example

Here's a specification of a Java method.

```
public double grossPay ( int hours, double rate  )
/* Calculate and return employee's gross pay.
 *   'hours' is the number of hours worked.
 *   'rate' is the employee's hourly wage.
 * Precondition: 1 ≤ hours ≤ 40 and 8.25 ≤ rate ≤ 15.00.
 * Postcondition: return value = hours * rate.
 */
```

Example

Here are the method specifications for the date validation program.

```
public int [] convertToValidDate( String date )
   throws IllegalArgumentException
/* Convert 'date' to a date on the Gregorian calendar.
 *   'date' has form mm-dd-yyyy or mm/dd/yyyy.
 * Precondition: none.
 * Postcondition: returns int array holding the month,
 *   day and year in positions 0, 1 and 2, respectively.
 * Throws IllegalArgumentException:
 *   If 'date' is null
 *   If 'date' doesn't have the indicated form.
 *   If mm-dd-yyyy is not a true Gregorian date.
 */

public boolean isValidDateFormat( String date )
/* Return true if 'date' has form ##-##-#### or ##/##/####.
 * Precondition: 'date' is not null.
 * Postcondition: returns true iff date has valid format.
 */

public int [] extractMoDayYr( String date )
/* Extract from 'date' three integers representing a month,
 * a day and a year; return them in an array.
 * Precondition: 'date' has form mm-dd-yyyy or mm/dd/yyyy.
 * Postcondition: returns a[0]= m, a[1]= dd, a[2]= yyyy.
 */

public boolean isValidDate( int month, int day, int year )
/* Determine if month, day, year is a true Gregorian date.
 * Precondition: none.
 * Postcondition: returns true iff m/d/y is a real date.
 */
```

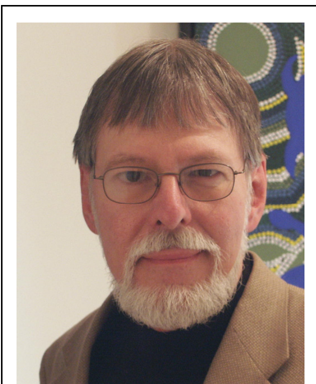
Structured Analysis and Structured Design

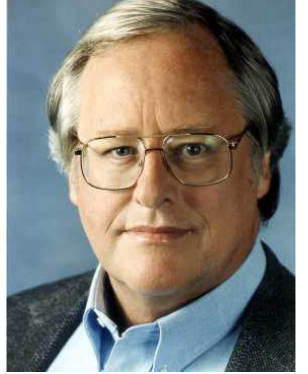
This topic is a very general overview of ***Structured Analysis and Structured Design*** (***SASD***), which is a software development model that depicts software as a hierarchy of tasks. SASD tools and methodologies were developed in the 1970s by many people. Notable are Wayne Stevens, Glenford Myers, and Larry Constantine who were early inventors of Structured Design.[1] Douglass Ross was one of the first to name and describe Structured Analysis.[2] Ed Yourdon and Tom DeMarco were very active in developing the details of SASD.

Wayne Stevens

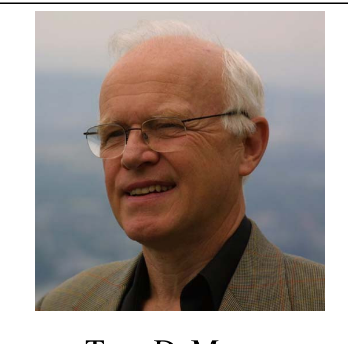
Glenford Myers

Unfortunately, since each of these men, and many others, wrote books describing different versions of SASD, [3456] no standard for it was ever developed. Thus, programmers wishing to use SASD have numerous diagramming tools and several design methodologies to choose from.


Larry Constantine


Ed Yourdon

In this topic, I have attempted to give a general description of SASD and extract those pieces of it that I have found most useful to novice computer programmers.


Tom DeMarco

[1] W. P. Stevens, G. J. Myers & L. L. Constantine, "Structured Design," *IBM Systems Journal*, 13:2 (1974), pp 115–139.

[2] D. Ross & K. Schoman, "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering*, 3:1 (January, 1977), pp 6–15.

[3] Tom DeMarco, *Structured Analysis and System Specification*, Prentice-Hall, 1979.

[4] Glenford J. Meyers, *Composite/Structured Design*, Van Nostrand Reinhold, 1978.

[5] W. P. Stevens, *Using Structured Design*, John Wiley & Sons, 1981.

[6] E. Yourdon & L. Constantine, *Structured Design*, Prentice-Hall, 1979.

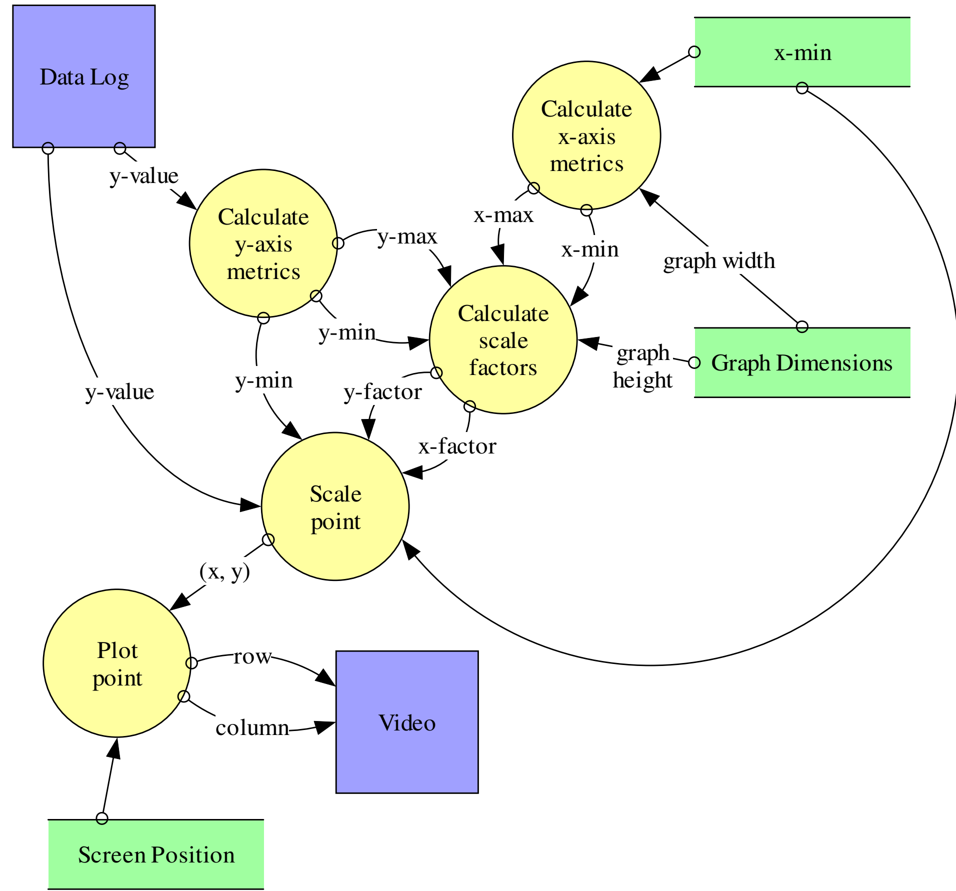
Exercises

1.	Draw a DFD that models using an automated fuel pump to gas your automobile. Assume that the only payment option allowed is using a credit or debit card at the pump.
2.	Draw a DFD that models checking out an item (e.g. book or video) from the public library. The librarian uses a gun to scan the patron's library card and an identification tag on the item. A receipt listing the item and its due date is emailed to the patron.
3.	Draw a DFD that models the spell check component of a word processor. The spell checker is given the document as a source and has an internal dictionary containing correctly spelled words. Each word in the document that is not in the dictionary is a suspected misspelling. It is displayed to the user along with a list of potential correct words. The user can (1) replace the suspect word within the document with one of the listed correct words or his or her own entry, (2) add the suspect word to the dictionary or (3) ignore the suspect word and leave it as is within the document.

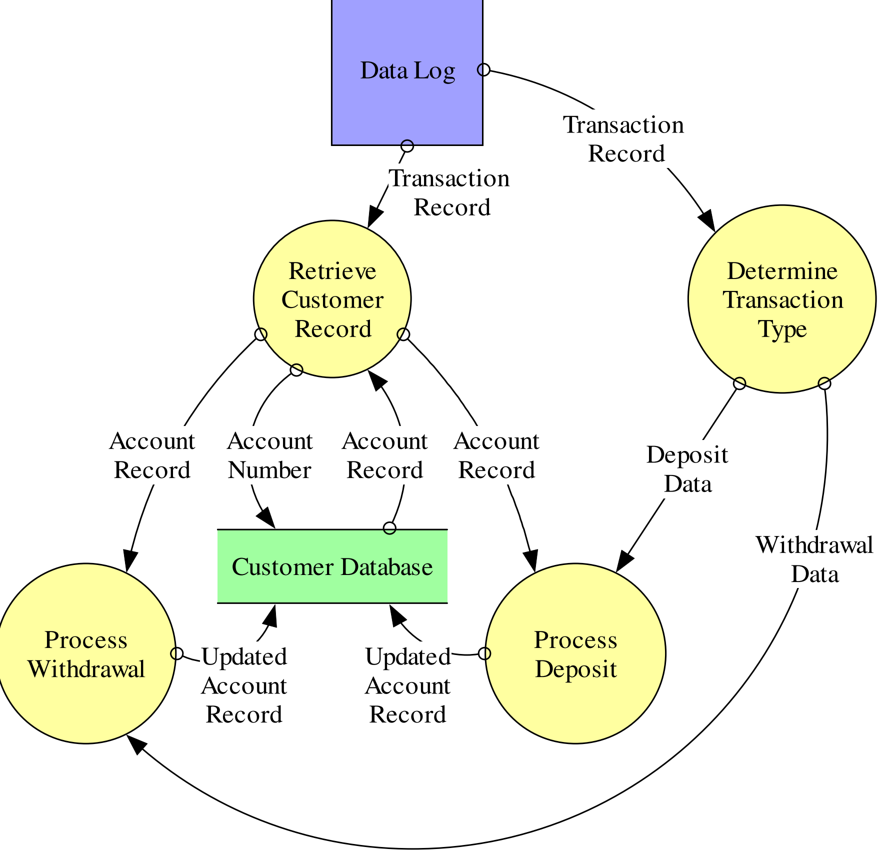
4. A data file contains a log of floating-point numbers. Each number represents a *y*-axis value and its position an *x*-axis value. For example:

Log:	0	100	50	75
Represents:	(10, 0)	(15, 100)	(20, 50)	(25, 75)

A program is needed that reads the log and displays the values as a graph on the computer's video screen. A DFD describing the process is shown below. Factor it into a structure chart.



5. A bank maintains a database of information about its customers' bank accounts. During the course of the day, all transactions made to a customer's account (through any means, Internet, ATM, phone, etc.) are recorded in a log file. At the end of the day, the log is used to update the database. A DFD describing the process is shown below. Factor it into a structure chart.



6. Write a method specification for the *Calculate Deductions* component of the *Calculate Net Pay* structure chart. Include pre- and postconditions.

7. Write a method specification for the *Calculate Income Tax Withholding* component of the *Calculate Net Pay* structure chart. Include pre- and postconditions.

8. Write a method specification for the *Calculate SS Tax Withholding* component of the *Calculate Net Pay* structure chart. Include pre- and postconditions.

9.	Write the specifications for the four methods in the median finding program. Include pre- and postconditions. Take into account the following additional requirements for *Input Scores*: a. The name of the CSV file is always **`gradeexport.csv`**. b. The CSV file contains one line for each student; the exact number of lines is unknown. c. Each line contains the name of a student (which may have embedded spaces), followed by a comma, followed by the assignment score (a floating-point value). d. If file **`gradeexport.csv`** does not exist, the code throws a **`FileNotFoundException`**. This shouldn't happen since the grade book interface part of the program creates the file. But, to aid debugging and to catch any catastrophic error during production runs, you must catch this exception and handle it by displaying an error message and terminating the program. e. If the contents of **`gradeexport.csv`** is not in the expected format, the code throws an **`InputMismatchException`**. This also shouldn't happen but you must catch it and handle it by displaying an error message and terminating the program.