Problem Solving Methodolgy

Most problems are poorly defined, complex, and basically unsolvable. A standard method for making problem solving **easier,** more **reliable,** and **more productive** is called **NAOMIE.** It provides a guidelines for application development.

NAOMIE - Needs(requests), Aims, Objectives (outcomes), Method, Implementation, Evaluation

This is a standard process for **designing** and **creating** solutions to problems. Consider the problem of keeping track of books in a library. A **systems analyst** would collect information about the library's **needs(requests)** - what are they doing with the books, how many are there, what information do they want to store about the books and the custormers, who are the customers, what happens when they take a book, etc. This preliminary study may include **sample data** such as:

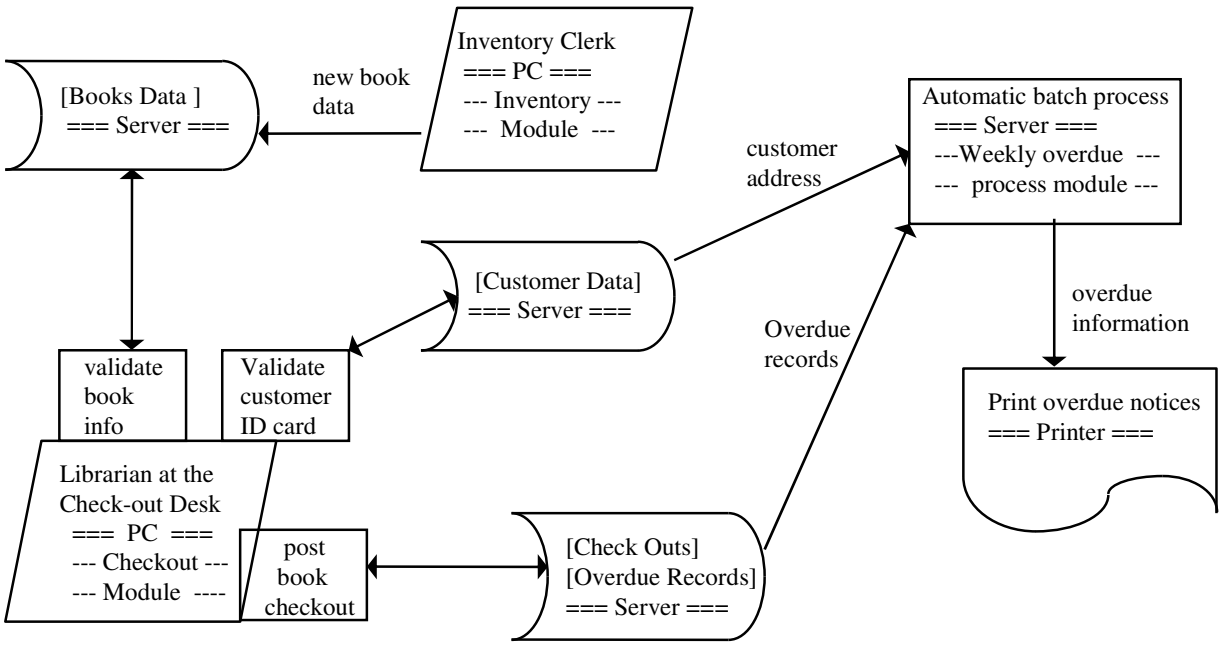
customer:	#123654, John Smith, 3450 Oak Str, 935-2312
book :	#123.e45, Gone With the Wind, Ernest Hemingway, 1935
magazine:	#1025.s25, Scientific American, June 1997
checkout:	#123.e45, #123654, 05.Jan.1997

Some needs/requests will be rejected, because the systems analyst decides any solution will be too expensive or difficult. For example, if the librarian wants to have constant up-to-date information about the whereabouts of the customers, so they can be tracked down at any moment. The remaining needs - the reasonable ones - become the **aims** of the project. These might be:

- keeping track of authors, titles, etc of all books
- keeping a list of names, addresses and phone numbers of all customers
- keeping track of which customrs have checked out which books
- automatic mailing of overdue notices

Objectives are more specific than aims, and include things like: total acceptable cost, desired hardware, number of workstations, total storage capacity, etc.

Methods include the **data-structures** (files, records, etc) and **algorithms** (procedures) which will be involved in the solution. The library problem may end up with one customer file, one book inventory file, one check-out file, and one overdue file, all stored on a hard-disk. Each **transaction** (a book checked out or in, a new customer, an overdue notice being mailed) requires **updates** in these files. The processes which must occur at each transaction should be written down. At this stage, an **IPO system flowchart** (Input-Process-Output) may be drawn, showing the **flow of data** between system **components.** This example **data-flow diagram** is not quite complete:



Implementation - A **specification document** must be written, detailing all the needs, aims, objectives, and methods. This can be a very long document. Indeed, it would probably be as long or longer than the programs to be written. This is written be a **systems engineer** who has more technical knowledge than the systems analyst. The **specification document** is given to the **programmers**, who must then **implement** the desired solution in a specific programming language.

Evaluation - After **programs** have been written and debugged, they must be **tested** thoroughly. **Evaluation** may result in the software or hardware being rejected because it is **too expensive, too slow, too unreliable, or too difficult to use.** For example, the original plan to have the librarian type customer numbers and book numbers into a keyboard may prove too difficult, slow, or unreliable. As a result, the system might be **redesigned** to use a light pen. Then new modules must be added which can print **machine readable** book labels and ID cards for the customers. These changes are **fed-back** to the systems analyst, who modifies his needs-assessment, and the engineer then modifies the specification document, and then the software and hardware engineers can start modifying their solution.

Software Design and Implementation

Software production includes **design** and **implementation** stages. As stated above, a **data-flow-diagram** is useful to get an overview of the **I/O and storage components** (devices and files) and the **flow of data** in the system.

Each **software module** (program or sub-program) should appear somewhere in the diagram, and is associated with a task, transaction, or process in the real-world.

Top-down-design and **step-wise refinement** are used to break down large, complex tasks into manageable sub-tasks. The resulting design can be displayed in a **hierarchical chart**, leading to a **pseudocode representation** of the **algorithm** used to solve the problem. Generally, there is one box in the hierarchical chart for each procedure, and the procedure is written in pseudocode, and this is done **before coding** the solution in a **higher-level language**, such as **Pascal** or **C++** or **BASIC.** A very thorough chart would include **parameter passing** along the lines connecting the procedures (not shown below).

A thorough **error discussion** (including **prevention, handling, recovery**, and **consequences**) is useful during the **design** stage. Early consideation of errors stimulates production of more **robust** software (reliable, maintainable, expandable). The error discussion includes **causes and prevention** (user-error, software bugs, hardware failure), **feasible exceptions** (irregular data), **means of recovery** (backups, redundancy), and **consequences** (cost of recovery, damage, destruction). This discussion can lead to **error-handlers** being designed into the system - **error prevention** and **detection** through **validation** and **verification -** as well as a complete set of **test-cases** (test data and expected results) for testing the finished software.

The sample chart below is for the **check-out-a-book** module. It is followed by pseudocode for some of the procedures. The reader is encouraged to produce an **error discussion** to go along with the documents shown below.

A good software design also includes **user-interface design** (forms and dialogues), **report designs** (e.g. lists of overdue books), and an **equipment list** of hardware which is expected to be available - possibly with approximate costs.

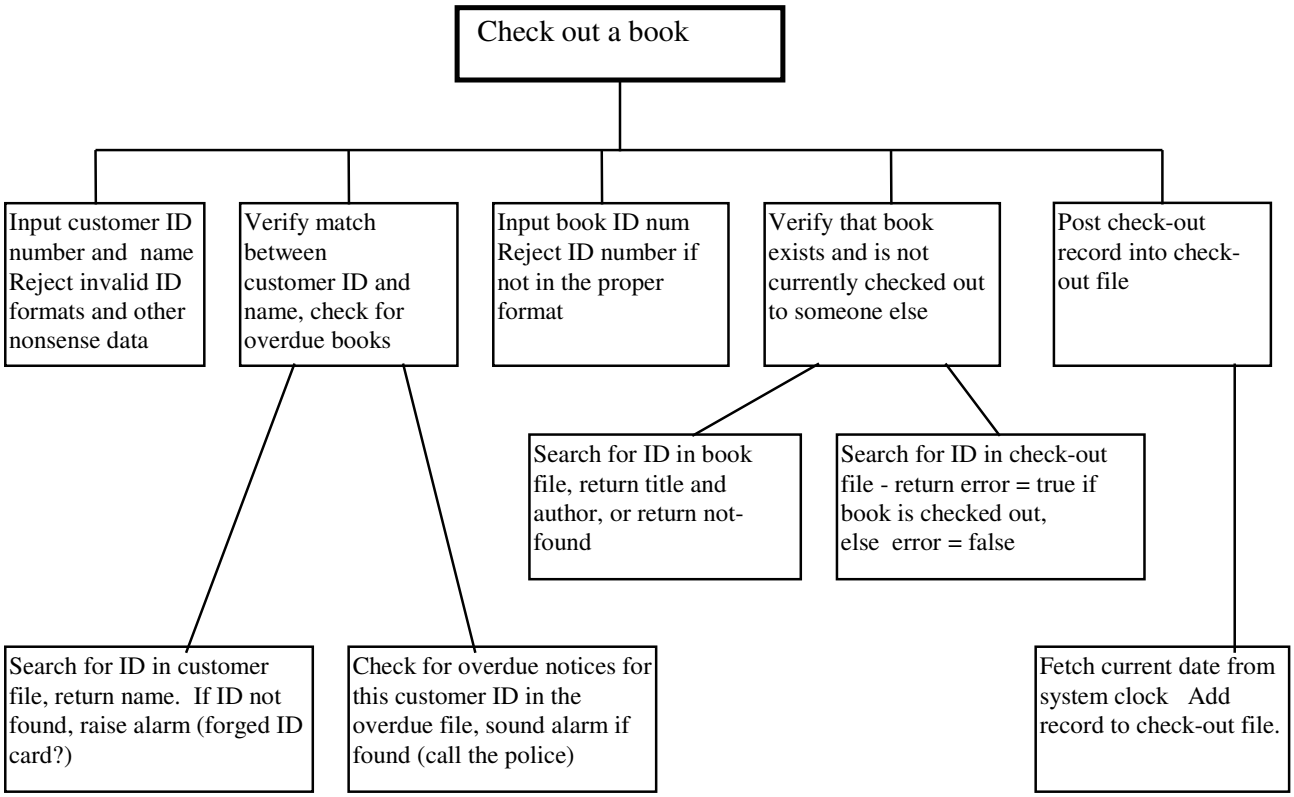
A really good design includes **alternatives** to some of the proposed components. For example, user input via **keyboard** vs **light-pen**, with pro's and cons.

Bottom-Up

Some software is designed and implemented from the bottom up. This means that programs are written without proper design and planning. This is also called **incremental developement** - a little bit of the program is written, then tested, then a bit more added, etc. This is the common method employed by beginners, who have not done enough programming to be able to think ahead before writing the program (some professionals, too!). This may be unavoidable, and all programmers have gone through the evolution from **bottom-up hacking** to **top-down design**. The bottom-up method seems to be more fun and feels like it is saving time, but in the long run it leads to a lot of false-starts and wasted time, and results in messy, limited solutions. The resulting programs are seldom **robust** (reliable, maintainable, expandable), and the programmers generally fail to meet development deadlines, as they cannot plan their time usefully.

Check-out-a-book Module

Top-Down-Design Hierachical Chart



==========

Comments (add to the error discussion):

The user (librarian) is expected to type the book-id number into the keyboard.
If the book-id is valid, the book title will be displayed on the monitor. The librarian **must** check that the displayed title correctly matches the physical book. If it does **not** match, the librarian **must** type NO, and the check-out process will be aborted. This physical verification should be more comfortable than the librarian typing the entire title into the computer, especially as many titles are very long and difficult to type. If the librarians are lazy about confirming the title (e.g. always typing "yes"), some mistyped ID numbers may result in incorrect bookings, and the entire inventory system will fail to function properly - incorrect overdue notices, books which cannot be checked out, etc.

Pseudocode

```
Procedure Check_out_a_book

      Error <-- Input_Customer(CustId, Name)
      if Error then exit            /* librarian might cancel with 0000 */

      Error <-- Verify_Customer(CustId, Name)
      If Error then
            display "Customer ID does not match name"
            exit
      endif

      Error <-- Check_Overdue(CustId)
      If Error then
            display "Customer has overdue books, check-out not permitted"
            exit
      endif

      Error <-- Input_Book(BookID)
      If Error then exit            /* cancel with 0000 */

      Error <-- GetTitle( BookID, Title )
      If Error then
            display "Title and BookID don't match or don't exist"
            exit
      endif

      Error <-- CheckedOut(BookID)
      If Error then
            display "Book is already checked out - cannot check out again"
            exit
      endif

      Error <-- PostCheckOut(BookID)
      If Error then
            display "Check-out failed - non-specific error"
            exit
      endif
endProcedure Check_out_a_book

Function GetTitle(BookID by-reference, Title by-reference)
      Open(BooksFile)
      Title="***"
      while not eof(BooksFile) and Title = "***" do
            Read(BooksFile) BookData
            if  BookData.ID = SearchID then Title <-- BookData.Title
      endwhile
      Close(BooksFile)
      If Title="***" then
            GetTitle <-- True       /* Error, book not found */
       else
            GetTitle <-- False            /* Okay, book was found */
      endif
endFunction GetTitle
```

Practice for Students:

(1) Write down some ideas for the error discussion. Concentrate on **feasible exceptions** and **common problems** - these are things which are likely to go wrong on a regular basis. For example, a **power failure** is not a common problem in Germany, although it may be a very common problem in another country. If power failures are frequent, then the software and/or hardware may be designed to minimize the catastrophic consequences. For example, it may be necessary to always keep a complete list of customer names and ID numbers, so that books can still be checked out when the power is off. But in Germany, we wouldn't spend a lot of time thinking about this problem. If you think that stolen or counterfeit ID cards are a real problem, list that. But perhaps this is not a realistic problem. Some causes of errors are very common and frequent. What are they? How "dangerous" are they? How can they be **detected** and **prevented?**

(2) The example pseudo-code does not contain any errors (hopefully). Choose one of the other procedures listed in the top-down-chart and write the pseudo-code for that procedure, similar to what was done for the GetTitle procedure.

(3) In the **system-flow-chart** (data-flow diagram) there is no indication of how data should get **into** the customer file. There will be new customers sometime. Who should enter the new data? How? What precautions should be taken? What errors should be prevented? What steps should the operator follow?

(4) In the **hierarchical chart**, write in the **parameters** which are passed along the lines between procedures. Include **arrows** to show whether the parameter is passes into the procedure (**pass by value**) or out of the procedure or both (**pass by reference**).

(5) The **hierarchical chart** (top-down-design) for the check-out process assumes a traditional **TTY** (teletype) style interface - **DOS** text mode, with questions and answers, following in a specific order from beginning to end. In **Windows** and other **GUI** (Graphical User Interface) environments, programs are more likely to be **event-driven** - this means that the user controls the order of execution by pushing buttons. Rather than question-answer-question-answer..., the programmer designs **forms** and **dialog boxes**, where the user types in their data (name, id, etc). First, they type in **all** of their data into several different **fields.** Then they use a **mouse** to press a **button** on the **form**. That **event** activates the program which processes **all** of the data. This means that the program sits idle while the user types lots of data. Then all the **validation and verification** occur afterward.

Design the input **form** for the data required for the check-out-a-book process, assuming a **GUI** style program. Be sure to include all the required **input fields**, **output fields**, and **buttons**, as well as any other useful elements. For example, would you display the current date on this form?

(6) Consider checking a book back in to the library. What information must be typed into the computer? How could data be **validated** or **verified**? Which data-files would be changed when the book is checked in? What errors could occur?

Draw a **hierarchical chart** to handle the task of checking a book in. Then write the **pseudo-code** for the check-in procedure. You do not need to write the pseudocode for any sub-programs in your chart - just the main procedure.

(7) Estimate the storage requirements for all the data-files, assuming the library contains 20,000 books, has 2,000 customers, and checks out an average of 40 books per hour, 5 days a week, 50 weeks per year.

(8) What **backup** procedures would you suggest, to prevent loss of data due to hardware/software **malfunctions**? That is, what should be backed up, how often, how should it be done, by whom, etc?

Object Oriented Programming

Studying the data-flow diagram for the library, you will observe a clear separation between **data** (stored on servers) and **processes** (rectangles). This is a traditional separation, connected to the idea that **programs** run in **RAM,** while data resides on **storage devices** (disk drives and tapes). In tradition high-level, block-structured programming languages, we also see this separation - **data-structures** such as arrays, linked-lists, and records are quite separate from **algorithms** which appear in procedures and functions.

Objects provide a package which contains both data (**properties**), and algorithms (**methods**) which operate on the contents of the object. By packaging the algorithms and data-structures together in a single module, **side-effects** are minimized, because algorithms are always operating on local data, never on global data. This improves **reliability**. Windows95 and other large programming projects are written using extensive **object-oriented** techniques. Such large projects benefit greatly from the improved modularity and reliability offered by **OO** techniques. Smaller projects (e.g. an IB portfolio) probably do not benefit from OO.

Consider our library case-study. Books get checked in and out, magazines are ordered and received, some books are reference books and cannot be checked out, and there are a number of other publications such as newpapers and CD's. Rather than treating each **type** of material separately, the OO programmer builds a **hierarchy** of objects. Start with the most general concept - a **publication**. Every publication has a title, author, date, a publisher, and a price. The title, author, publisher, and price can be set directly. However, to prevent bad data from getting into the system, the date cannot be simply recorded as a string - this is done through the **set_date** method, which does some error-checking before recording the date. By making the date field **hidden**, all algorithms are required to use the error-checked procedures to set these values, thus increasing reliability. Now **publication** becomes the basis for all other objects - books, magazines, newpapers, CD-ROMS, etc. The **hierarchy** below shows the methods and properties of various publications. These are **classes** - these describe **types** of objects. The actual objects are declared just like variables are declared from predefined types.

```
Publication {-------------------------------------------------------------------------
    |                   properties:
    |                         public:  title, author, publisher, price
    |                         hidden: date
    |                   methods:
    |                         set_date, get_date
    |              -------------------------------------------------------------------------}
    |_____
    |       Book    {-----------------------------------------------------------
    |                   properties:
    |                         public:   pages, publisher, abstract, ID (Dewey Decimal Number)
    |                         private:  isbn, due_date, customer_ID
    |                   methods:
    |                         set_isbn, get_isbn, check_out, get_due_date, get_customer_ID
    |                 -----------------------------------------------------------}
    |_____
    |       Magazine {-----------------------------------------------------------
    |                   properties:
    |                         public:  volume, number, contents, ID (Magazine id code)
    |                         private:  renewal_date, due_date, customer_ID
    |                   methods:
    |                         check_out, set_renewal_date, get_renewal_date
    |                -----------------------------------------------------------}
    |_____
            Reference {-----------------------------------------------------------
                      properties:
                            public: volumes, abstract, ID (Dewey Decimal Number)
                                                  /* cannot check-out a reference book*/
                    -----------------------------------------------------------}
```

Inheritance

The types **Book, Magazine,** and **Reference** are **derived** from the **base-class Publication.** Through the process called **inheritance**, the child types **automatically inherit** the properties and methods from the **Publication** class. Thus, the new properties and methods of the **Book** class simply extend the **Publication** class. This has several advantages for programmers.

(1) Base classes and entire base hierarchies can be created and stored in a library. Then the application programmer can take this library and easily extend it. There is no need to start over again from scratch.
(2) If a large programming project is underway, and someone decides that the set_date method should be improved, this can be done and all the improvements are immediately transferred to the derived classes, without any need to rework them.
(3) Inheritance can continue through any number of levels in a hierarchy. At the end, the application programmer need only deal with the highest level descendents, and needn't worry about technical details of the base class. For example, it might be sensible for the magazine object to override the get_date method, to perform a tighter range-check on the values.

Traditional languages allow a form of inheritance by creating larger data-structures from simpler data-structures, but this only provides inheritance for data-structures, not for algorithms.

Polymorphism

Polymorphism means that the same name can be used to mean different things for different objects. The check_out method is different for books and magazines - the magazine doesn't get checked out as long, and different information needs to be typed in by the librarian. Nevertheless, the name **check_out** can be used for the two different methods for books and magazines. As a result, the same code can be used in the main program to manage the check-out module, without having a bunch of if..then.. commands to call different functions for different publications.

The concept of polymorphism exists in traditional programming languages. For example, the operator + can be used in Pascal to add two numbers, or to concatenate two strings. Thus, the + operator is polymorphic. The **writeln** procedure is also polymorphic - it will do something quite different with strings than with numbers, and can write either onto the screen or into a file. And different records can have fields with the same names but different meanings.

Many OO languages allow methods or properties to be redifened in child classes, thus **overriding** a method or property from the base (parent) class.

Encapsulation

Encapsulation refers to several aspects of objects:
(1) Algorithms and data-structures are encapsulated into a single package, which can be manipulated as a whole - copied, created, destroyed, etc.
(2) Some or all of the data can be **protected** from outside interference. In the example above, the date can only be accessed through get_date and set_date. This guarantees that programmers don't bypass the proper error-checking code which is contained in these methods, and also guarantees that there will never be any bad data in the date field. This sounds silly to single programmers, but in a large project this is a very useful precaution to prevent one programmers modules from interfering with anothers'.

The concept of protection occurs in traditional languages, where **local variables** are accessible only to the procedure where they are declared. When the procedure is copied or used in a new program, the local variables and parameters go along and are protected from corruption by the new environment.

Advantages of OO

Polymorphism is possible in traditional languages, but is easier with objects. Encapsulation is possible without objects, but most traditional languages do not allow protection of variables. Inheritance of methods is a feature of OO which is new. It is a fundamental theorem of computer science (Böhm-Jacopini, 1964) that every program which can be executed on a computer can be written using only input, output, loops, if..then.., and variables. That means that all traditional languages can be used to write exactly the same programs as OO languages. Indeed, OO programs must be **compiled** to machine language just the same as traditional languages. So OO is not magic. The advantages of OO are **convenience** and **productivity**. Encapsulation provides protection, inheritance simplifies code, and polymorphism improves readability and reusability. Altogether, these **increase reliability** and **speed debugging** - two major issues for large programming projects - which explains the popularity of OO among professional programmers.

Cafeteria Design

Our school cafeteria uses a computer system to track payments for food. It basically works as follows:

Each customer (student and teacher) receives an RFID tagged "chip". The customer must **pre-pay** money into their account. Each time they buy something:

- the cashier types in codes for the purchased item(s)
- the computer system checks for the correct prices and adds them up
- the total price is displayed
- the customer places their chip on a reading device
- if there is enough money in the pre-paid account to cover the purchase, then a green light flashes and the purchase is charged to that account - then the new account balance is displayed
- if the account does not have enough money to cover the purchase, a red light flashes. Now the customer can walk to a cash collection machine, insert cash into the machine, and then return to the cashier to pay.
- in an emergency (e.g. a visitor or a lost or forgotten chip), customers can pay cash directly to the cashier, but this is infrequent.

1. Outline 3 advantages of using the computerized chip system for payment - these must be advantages for the cafeteria staff, not advantages for the customers.

2. Compare the use of RFID to 2 alternative computerized ID systems. You must clearly identify the alternative systems (not cash).

3. The cafeteria chip system requires an Internet connection. Suggest why the Internet connection is required.

4. Discuss whether the cafeteria chip payment system should be connected to an inventory system. Whether your answer is yes or no, supply 2 supporting arguments.

5. Draw a **system flowchart** for the chip payment system, including **data-flow** between modules.

6. Assume that the ID codes of all the customers and their current balance are stored in 2 parallel arrays.
   (a) Write a method for the machine that accepts cash and updates the customer account.
   (b) Explain why this method (and data structures) cannot simply exist in the cash entry machine.

7. HL
   (a) Outline a suitable OOP class hierarchy for the general class MACHINE and the more specific classes PAYMENT_REGISTER and CASH_COLLECTION.
   (b) Describe a DATA STRUCTURE that should be contained in the MACHINE class.
   (c) Describe a METHOD that would only be present in the PAYMENT_REGISTER class.
   (d) Describe a method that should be available in all 3 classes.

8. Describe one advantage and one disadvantage of allowing chips to be used without the need for photographic identification or passwords.

9. Describe a social issue connected to the use of the chip payment system.

10. Suggest an alternative computerized payment system that does not require the existence of chips.