IB Higher Level Computer Science Dossier

2004

v4.1

TABLE OF CONTENTS

1. INTRO

The Program Dossier is an individual piece of well-documented work completed during the course involving a problem that can be solved using computer systems. The emphasis is on the use of a logical approach and analytical thinking from definition and decomposition of the problem through to its solution by constructing algorithms in a high-level modular programming language. This can be carried out in a procedure- oriented or an object-oriented environment.

The Program Dossier is internally assessed by the teacher and externally moderated by the IBO.

Choice of Problem

- Candidates are free to choose problems generated by themselves or their teacher. They can take full advantage of local resources and circumstances, but must make choices that allow the mastery of the objectives to be demonstrated. Candidates may share the same problem to be solved or the same initial scenario, but collaborative work is forbidden.
- Teachers are expected to give educational guidance at each stage of the design process, **but the Program Dossier of submitted work must be that of the candidate alone.**
- Some dossier ideas from Shashi Krishna (teacher in IB OCC discussion thread 24/03/2003):
  1. CAS hour database - every IB school has the CAS hour requirement and so this would be in-house and getting information would be easy.
  2. Lab Inventory System for keeping track of machines, labs and machine info.
  3. College Application Database - useful for high school counselors who want to keep track of the students applications and college info.

Time Allocation

- It is expected that approximately 35 hours teacher contact time will be devoted to the Program Dossier, including guidance on format, presentation and contents. Some of the time teaching the syllabus content will also involve work connected with the Program Dossier, but **this does not include time required by students to work on their own** to develop and complete their dossiers

Weighting

- The program dossier is worth **35%** of the final grade.
- +Paper 1 is worth 30% + Paper 2 is worth 35%= 100%

Teacher Assessment

- Teachers judge candidate's performance by using level descriptors against the relevant criteria which are related to the objectives. The criteria and achievement levels must be applied to the work in the Program Dossier regardless of the number of aspects in which mastery is demonstrated. After this a 'mastery factor' is applied which depends on the number of different aspects in which mastery is demonstrated. The assessment of the Program Dossier is moderated externally.

- Only the code designed and written by the candidate must be taken into account when applying the assessment criteria.

- If the Program Dossier does contain two independent programs then the final marks submitted should be a holistic judgement reflecting the Program Dossier as a whole, not an average of the work contained in it. It is therefore essential that both programs should be fully documented.

PASCAL

- "Pascal could be the language of choice for those who don't want to take risks with their students – PURE and Pascal are very close therefore students don't have such a gulf between programming theory papers" [IBO email thread Nov 16, 2001 Richard Jones, UWCSEA Singapore]
- "I am teaching C++ at the moment and used to teach Pascal. These are pretty similar, but C++ does let the careless programmer get into more trouble at times, - for instance, it allows a program to write beyond the end of any array, where the version of Pascal I used would flag the problem with an error message." [IBO email thread Nov 17, 2001 Jack McCormack

DOCUMENTATION

- "A typical, good quality, Higher Level dossier will have **10-15 pages** of documentation (not including the program listing and sample runs" [Deputy Chief Examiner Glenn Martin, Oct. 24, 2000]

FORMAT [p. 54]

- All the candidate's work must be submitted together as a single document. The work can be stapled or put into a ring-binder. All information required for the Program Dossier must appear as hard copy. Diskettes, CD-ROMS, etc. must NOT be included within the Program Dossier or sent to the moderator.
- There MUST be a **Table of Contents** and all written documentation should be word-processed, except where it is felt necessary to include rough notes.
- The pages of program listings and sample runs must be separated AND labelled and not submitted as continuous sheets.
- All the **pages must be numbered**. The numbering can be sequential (1, 2, 3, etc.) throughout the entire Program Dossier or it can be done according to the sections (**A-1, A-2, A-3, B-1, B-2, etc**). This may be easier, since each item can be numbered sequentially as it is completed. The page numbering can be done by hand, but NOT recommended!
- The number of pages associated with each item may vary according to the nature and complexity of the problem being solved as well as its programmed solution. However, as a guideline, an approximate number of pages is given (by the IB) in the following table.
- All of the items listed the table must be included in the Program Dossier

ITEMS TO BE INCLUDED		NUMBER OF PAGES (APPROX)	CRITERIA TO WHICH THE ITEMS RELATE
1	TABLE OF CONTENTS	1	
2	STATEMENT AND ANALYSIS OF THE PROBLEM	3	A
3	DESIGN PROCESS DOCUMENTATION	4 MIN	B, D, I
4	DATA STRUCTURES	2	C
5	TESTING STRATEGY	3	E
6	EVIDENCE OF USER FRIENDLINESS	3	G
7	ERROR HANDLING FACILITIES	3	H
8	HARD COPY OF TESTING	30 MAX	F, G, H, I
9	PROGRAM LISTING	-	D, G, H, I, J
10	USER DOCUMENTATION	2-3	L
11	CONCLUSIONS/EVALUATION	1	K

53 pages total

2. MASTERY FACTOR

- "Candidates have to demonstrate mastery of various aspects of their chosen programming language by documented evidence consisting of **one program** (but not more than two independent programs) that addresses a single problem. Mastery is defined as **the ability to use an aspect appropriately for some non-trivial purpose which is well documented.**"

- The candidate must attempt to show mastery of **at least nine** of the following aspects [but you will attempt all of them].

CALCULATION OF MASTERY FACTORS	
9, 10 or 11	**x 1.00**
7 or 8	**x 0.75**
5 or 6	**x 0.50**
0, 1, 2, 3, or 4	**x 0.25**

The first three go together:

1. **INSERTING a new data item into a LINKED LIST or TREE**
   Note:
   - It is not sufficient to add a data item only to the front or only to the rear of a list

2. **DELETING a data item from a LINKED LIST or TREE**
   Note:
   - It is not sufficient to delete a data item from only the front or from only the rear of a list.
   - There must also be proper disposal of the allocated memory (for example, updating the free space pool in an array implementation, disposing of a node in a pointer implementation).

3. **SEARCHING for a data item in a LINKED LIST or TREE**

"Candidates must attempt to show mastery of the following aspects:

- adding directly a new record to a file
- deleting directly a record from a file
- searching directly for a record in a file.

To satisfy these requirements, candidates must carry out file manipulation in such a way that files **do not need to be read into RAM for manipulation.**
[Computer Science: Clarification of Internal Assessment Details – February 2000]

4. ADDING DIRECTLY a record to a file

Note:

- "Do not read it into an array, alter it and write it back again, as this is not direct (random) access." Richard Jones, Singapore
- "Either last year or the year before, a clarification on this was sent out to all schools because there was (still is) confusion about what must be done to demonstrate mastery when adding or deleting records "directly" from a file. It is intended that when adding a new record, the record will be added via main memory, then an appropriate place for it will be found in the file (stored in secondary memory where the record is then stored). On the other hand, the entire file should not be loaded into main memory (i.e. as a linked list or tree) where the new record is added as a node and then the entire file is rewritten to disk. [email from Jack McCormack, March 27, 2002]
- The file manipulation must be done "in such a way that files do not need to be read into RAM for manipulation." In other words, "When directly adding new records to a file, it is not acceptable to read the file into RAM, add the data, and rewrite the file".
  [Computer Science: Clarification of Internal Assessment Details – February 2000]

Summary
To meet the requirements of the Mastery of Aspects for Higher Level, direct manipulation of files must be carried out in such a way that the whole file does not need to be read into RAM.

5. DELETING DIRECTLY a record from a file

- "The record to be deleted must be found directly by searching the file in its actual location (by, for example, using a binary or linear search, reading a record, or small block of records, at once). The record could then be directly deleted (if the programming language supports this), or marked for deletion by using a flag field, or by using a rogue value in the key field. However, it is not acceptable, for example, to delete a record from a file, to read the whole file into a linked list or tree, to delete the node, and then to write the data back to the file. (However, this manipulation would satisfy the requirement of being able to delete a data item from a linked list or tree.) Provided the file size alters, or a record marked for deletion is actually overwritten during the addition of new records, this manipulation would meet the requirement.
  [Computer Science: Clarification of Internal Assessment Details – February 2000]
- " In the case of text files, mastery would depend upon whether a candidate has, for example, removed a line (or other chunk of data representing a discrete record) from the text file and either marked that as a space that could be reused in a subsequent add operation, or shuffled following records over it. That, in turn, would depend upon whether the file is maintained with serial or sequential organization. Either way, the data in the file is manipulated directly (**not**, for example, read into an array, altered and written back). "
  [Diploma Programme Coordinator Notes, February 2002]

6. SEARCHING DIRECTLY for a record in a file

Notes from email discussion threads on ADDING, DELETING AND SEARCHING DIRECTLY:

- *"The term* ***"direct-access"*** *is not very common in the literature - most books call this* ***"random-access"****. So when looking for a book, look for random-access files.*

  "In several languages, the command that implements the "random" part is called ***"seek"*** *(or "fseek" in C). It is the use of the seek command which distinguishes random access files. In PURE, this is the* ***"moveto"*** *command. So I would look for "seek" in a book index. And I think HL students should learn to use the "seek" command (or it's equivalent).*

  "Various languages use a large variety of commands for "opening" the files to enable random-access - in C++, these could be opened in "binary" mode (I believe that is correct, but I'm not really a C++ expert). In Visual Basic, the files are opened for "random" access, and in Pascal it would be a ***"file of recordtype".***

  "In the past, languages like ***Pascal provided read/write constructs for "records", and these would normally be stored in random-access or binary files****. This concept has disappeared in modern OO languages like Java, where you can only read/write single fields - there is no "record" construct. So if you look for "records" in the literature, you will probably only find useful stuff for older languages, like Pascal and Basic.*

  "Modern languages and introductory textbooks don't spend much time on random-access files - streams have become very popular, especially for distributed computing. The growth of distributed computing models seems to have displaced the more traditional local file concept, so there really isn't much about random-access files in modern text-books.

  "I just started using Java in my IB course last year and spent several months searching for an appropriate text-book. One thing I looked for was a chapter on random-access files, but never found that in any book. The best book I found was "Computing Concepts with Java Essentials", by Cay Horstmannm. He devotes about 25 pages to streams, and 5 pages to the RandomAccessFile class. This seems to be the trend in CS1 textbooks these days.

  "I believe a note in one of the IBO publications a couple years ago said something like ***"use of hashing in direct-access files is sufficient to demonstrate mastery"****. But I don't have the exact quotation handy - perhaps someone else can provide the exact statement. In any case, hashing algorithms are covered in depth in many text-books, but not necessarily in relation to files. Some notes on this specific topic are available at :*
  http://www.ib-computing.com/Files_HL.htm .

  "I will write my own notes about random-access. I will keep looking for good resources - if I find anything, I'll post it.
  [Dave Mulkey, 31/08/2003, OCC Computer Science discussion thread,
  http://web3.ibo.org/ibis/occ/fusetalk2/forum/messageview.cfm?catid=19&threadid=3810]

- *" There is no restriction regarding text or binary files of data. However…, random access is clearly a requirement."*

7. **MERGING TWO SORTED DATA STRUCTURES**
   Note:
   - "I would recommend Quick-Sorting the files and then merging them. This would definitely cover both the merging AND the recursion mastery factors" Richard Jones, Singapore

8. **USER-DEFINED FUNCTIONS**
   Note:
   - In this context a function is defined as a subprogram that evaluates to a single value (a returned value)
   - The implication of a "user-defined subprogram would be that it would need to be a function and thus return a value otherwise it would be just another procedure". Richard Jones, Singapore

9. **USING ARRAYS, RECORDS, OR POINTERS AS PARAMETERS**
   Note:
   "The main weakness I have seen in some dossiers is the trivial use of parameters. I have seen some programs with 10-20 procedures and functions where parameters were only passed in one instance and global variables were used everywhere else. This does not demonstrate mastery of using parameters." Jack McCormack

10. **RECURSION**
    Note:
    - Example: a **quick sort** is a non-trivial use of recursion
    - Example: "**tree-traversals** are non-trivial uses of recursion (since removing recursion would involve setting up your own stack instead of using the system stack)." Rod Uveges
    - Example: "A candidate that, say, used recursion to **print a linked list of ordered search results in reverse order could be credited with mastery if they have discussed other ways of carrying out this task and thus justified the recursive approach**. However, as a teacher, I would urge my students to do more than this just to be on the safe side." Richard Jones, Singapore

11. **Using more than one kind of COMPOSITE DATA STRUCTURE or using one hierarchical composite data structure**
    Note:
    - A **composite data structure** is one made from other data types. An example is record.
    - A **hierarchical composite data structure** is one which contains more than one element and at least one of the elements in a composite data structure. (Example: "An **array of records** is considered to be a hierarchical composite data structure because an array is one composite data structure containing elements (records) which are also composite data structures", says Jack McCormack). "A file of records is **not** a hierarchical composite data structure" (Dave_Mulkey@fis.edu. Jan 4, 2001)
    - Another example is a **record which has one field defined on another record** or **array**
    - "Examiners have again been discussing the issue of composite data structures and have decided that where candidates use two different composite structures (**example an array of records and a linked list of records**) that also fulfils the criterion." (Jack McCormack, Oct 24, 2000)

DYNAMIC DATA STRUCTURES AND POINTERS [p44 Assessment Details]

- Your program **must include linked lists or trees**
- Most candidates will probably use pointers to implement these linked data structures. Consequently, most students will use a programming language which supports pointers (ie. PASCAL). Although HL students have to learn to use pointers as part of the syllabus, the demonstration of Mastery of pointers in the Program Dossier is NOT required.
- Your program must properly **DISPOSE** of de-allocated memory by returning it to a free memory pool. For example, if the program does not properly dispose of de-allocated memory then the student would not be considered to have shown mastery of deleting an item of data.
- Obviously, a student who has used pointers in the Dossier will probably be better prepared for the exam than one who has not.

You must create a MASTERY FACTOR INDEX which includes the following:

A) A list of the 11 mastery factors
B) A description of how you will use demonstrate each factor
C) Indicate exactly where you will each factor (so the examiner can find it easily – if he/she can't find it easily no credit will be given for that factor)
D) Indicate exactly where the *explanation* for using the mastery factor can be found (in C-Data Structures and/or D-Algorithms)

- Include page reference numbers in the program and in the documentation!

This should be done in the form of a chart, for example this one:

Mastery Factor	**How I will demonstrate this Mastery Factor**	**Page no(s) of Where in my program to find this Mastery Factor**	**Page no(s) where the Explanation for using this Mastery Factor can be found on page..**
1. Inserting new data item into a linked list			
2. Deleting a data item from a linked list			
3. Searching for a data item in a linked list			
4. Adding directly a new record to a file			
5. Deleting directly a record from a file			
6. Searching directly for a record in a file			
7. Merging two sorted data structures			
8. User-defined functions			
9. Using arrays, records and pointers as parameters			
10. Recursion			
11. Using more than one kind of record (composite data structure) AND array of records (hierarchical composite data structure)			

3. ASSESSMENT CRITERIA

A. Analysis of the Problem [3 marks]

ASSESSMENT GUIDELINES for Analyzing the Problem: [p. 55]

"This should include a brief statement of the problem as seen by the end-user. A discussion of the problem from the end-user's point of view should take place with both the user's input and desired output being considered.

" The analysis should state a clear understanding of the problem and demonstrate how the problem can be solved using a computer-based solution. This analysis should also take into account what input and output will occur and what calculations and/or processes will be necessary to obtain the desired output."

ACHIEVEMENT LEVEL for Analyzing the Problem: [p. 47]

*"The documentation should contain a thorough discussion and analysis of the problem which is being solved. This should concentrate on the **problem** and the goals which are being set, not on the method of solution. A good analysis includes Sample data, Information and requests from the intended user, and some background of how the problem has been solved in the past".*

3/3: "The candidate analyses the problem to be solved"

Notes from Chief Examiner Glenn Martin, Bahrain, 2000
The examiner will look for five things:

1. A clear description of the problem
2. Sample data
3. Information and requests that the user will want
4. Background of how the problem has been solved in the past
5. The objectives

Note1: Don't mention the word "computer" here.
Note2: Don't mention array, data structures, etc. here
Note3: Place any interviews in an Appendix, and just summarize the interviews here.

Notes from Richard Jones, Singapore, 1999
In this section you need to state clearly what you are trying to solve. You must not fall into the trap of mentioning the **solution**! At this stage you will not have clearly thought through the analysis of the problem to even consider the solution. Avoid writing statements such as 'I have decided to use....'. This clearly says that you know what the solution is.

It is much better to give a brief introduction to the area in which you are working and then state the specific problems.

Use phrases such as ....

- 'The supermarket in question found difficulty in ordering the right quantity of perishable goods for the day.'
- 'The doctors at the surgery realised they were not running an efficient scanning process because they lacked the necessary information about their patients.'

A.
(i) Description of Existing System

- **A clear DESCRIPTION** (the facts only, no opinions) of the existing system including:
  - volume of data
  - **Sample data** and what is done with it.
  - Data Collection→ Data Preparation → Data Input → Data Processing → Information Output
  - **Typical information and requests the user will want**
  - how a new record is added, deleted, modified
  - how to search for a file, record
  - how the records, files are sorted
  - where/how records, files are stored
  - data security
  - data integrity
  - data validation
  - Place your interview and any questionnaires in the Appendix. Summarize the interview and any questionnaires in this section.
  - A data flow diagram could be included here (see Bradley p. 308)

Data Flow in Existing System: (Richard Jones, Singpore 1999)
'A picture is worth a thousand words' is an old proverb. Perhaps the best way to explain the data flow though a system is to draw a diagram which is fully labelled. You will then find it is much easier to explain what is happening at present within the current system. Show all data sources, both **external** and **internal**. Show also how they pass into the system and onto each stage before passing out of it. Point out any areas where likely problems might occur. Don't forget that the current system may not even begin to address part or perhaps any of the requirements. However you must still show what is happening.

(ii) Analysis of Existing System:

- An ANALYSIS of the issues you described above (now you can include your opinions).
- <u>Explain</u> all the things that should remain the same / the things that are good about the existing system example: The format of the data capture form should contain the same fields.
- <u>Explain each</u> of the problems related to this existing system :
  - ❑ Ex. Explain possible data capture errors (if there are any)
  - ❑ Ex. If it's too time consuming and inefficient, explain every part of the existing that is time consuming and inefficient.
  - ❑ Ex. If it's insecure, explain in detail why
  - ❑ Ex. If data is redundant, explain what data is repeated or typed more than once.
  - ❑ Ex. If there is a probability of error, list what types of errors and how they could occur. Analyze data validation.
  - ❑ Ex. If too much space is used to store all the records, then explain this too.
  - ❑ Ex. Problems with searching? Explain
  - ❑ Ex. Problems with sorting? Explain
  - ❑ Ex. Is it expensive? Is money wasted? Explain
  - ❑ Ex. Is it harmful to the environment? (ex. Noise pollution, uses too much paper) Explain.

Notes from Richard Jones, Singapore, 1999:

- If the system is a manual one and computer solutions already exist then don't be afraid of saying so. [They may not do exactly what you wish them to or they may be too expensive to buy.] However they ought to be mentioned.
- It is easy to fall into the trap of listing solely bad parts of the existing solution. You should avoid this at all costs. Mention what is good and weak about the system in use. Don't forget to explain each point as this will help you when you form your solution. Good points can be kept or mirrored by your computer solution.

(ii) Background of how the problem has been solved in the past

- Do some research!! Find out if someone has already created a database that does what you need to do.
  Ex 1. A pre-made database called ___can be purchased from ___ for ___ $US. The web page address is ___
  Ex 2. An online database called ___ found at http://www._____ can be used.
- Describe the pre-made software

(iv) Analysis of How the Problem Has Been Solved in the Past

- Describe and then Analyze the pre-made software that is out there.
- Try to discount the use of pre-made software!

B. Aim & Objectives

- Overall aim of the new system
- A list of objectives of the New System

Notes from Richard Jones, Singpaore, 1999:
Here you should give a brief introductory statement and then write a list of the aims. In other words you must say exactly what the 'new' system should achieve for the users. Remember, you are still **not** trying to provide the **solution** yet! Remember there is a definite link between the specific problems describe in section 1 and the specific objectives here in section 2 !

Examples:

- 'No complaint should be left unanswered for more than 24 hours.'
- 'No drug should be given to which a patient is allergic.'
- 'Clear lists of goods needing to be reordered should be supplied at the end of each day.'

B. Design [6 marks]

ASSESSMENT GUIDELINE for Documenting the Design Process: [p. 55]

- "An illustration indicating the top-down design of the solution to the problem should be produced using pseudo-code or other suitable means candidates should create an algorithmic representation of their solution.
- The algorithms produced should be independent of the target programming language.
- A programmer using almost any high-level programming language should be able to generate a computer-based solution from the submitted algorithms.
- Thus, it is imperative that all algorithms/subalgorithms that are essential to the functioning of the resulting program be included."

ACHIEVEMENT LEVEL for Documenting the Design Process: [p. 48]
The solution to the problem should be thoroughly designed before any programs are written, and this design process must be documented. Good top-down design results in a flexible, general, extensible solution. In this category, both the quality of the resulting design and the design process are being evaluated. The design must include a detailed representation of the algorithms used (via pseudo-cod, structure diagrams, etc) that clearly illustrate the candidate's solution.

Top-Down analysis (solution decomposition) means breaking down a problem into smaller problems. These are then broken down in turn until ultimately a pseudo-code representation is obtained which can be used as a basis for a program construction. It is appropriate to use diagrams for the early stages. However for the non-standard or non-trivial module, the final stage MUST be pseudo-code at a level of detail equivalent to PURE [though you don't have to use PURE]. This final stage of design should lead easily into coding in an appropriate programming language. For example, an object-oriented design should be able to be coded into several object-oriented languages, whereas a procedure-oriented design should be able to be coded into any one of several block-structured languages.

This criterion refers to the documentation of the design process which does ***NOT include the final program listing****.*

6/6: "The candidate includes documentation for a design that is ***complete*** *and* ***portable****.*

complete:	*All the relevant decomposition from the problem definition through all stages to the final stage are included.*
portable:	*The final stage of the design can be coded into more than one appropriate modular language.*

Continued…

AUTOMATED DEVELOPMENT SYSTEMS AND LIBRARY MODULES [p44 Assessment Details]

- Some programming systems, such as visual systems, provide interactive development environments with a wide range of extra facilities, such as visual design, object manipulation, and automatic code generation. However, the use of these is beyond the scope of this syllabus.
- Within the Program Dossier such facilities **may** be used for minor task, such as **formatting a dialogue box**, but must not be used for more complex tasks.
- For example, you are expected to write your **own algorithms of sorting** an array, rather than simply executing a library function which sorts the array.
- You are expected to write your **own algorithms which maintain a linked data structure**, rather than using a system library which already contains all the required algorithms.
- Any program listing that includes code automatically generated by the development system must have this code clearly identified and distinguishable from the code written by the student.

Notes from Richard Jones, Singapore, 1999:

Overall Plan for the Development of the Solution:

- After an introduction stating the general method you will use to solve the problem you should detail the steps you will take. You will do this along with a rough guide as to the time you will need for each section.
- You need to consider a number of factors in the development of your system. Some of these factors will be dependent on others. In some cases you will not be able to proceed to a stage without successfully completing previous items.
- A good way of showing your plan is to draw a diagram. If, however, you think of an alternative method that is fine.

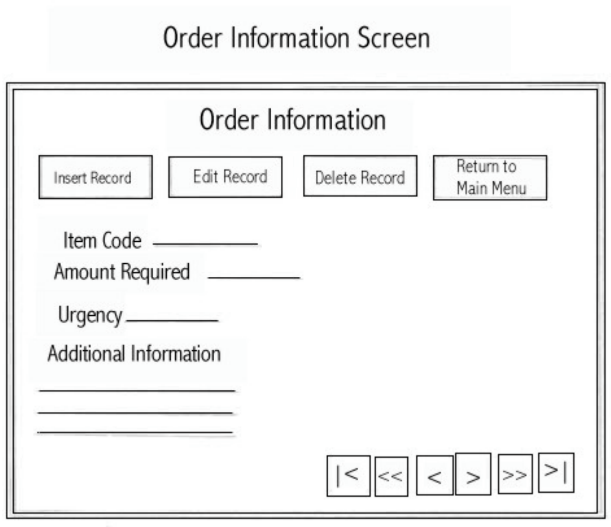
Hardware & Software Requirements

Explain the hardware and software your system will need to run. Mention particular machines and operating systems. You **must** also give reasons why you have chosen each device/software. Make these reasons applicable to your system!!

Outputs, Inputs, & Data Structures

- You need to consider what information your system will provide the users at each stage of the processing. Also you must think about what form it will take - screen, printer. This will help you to decide what information the users must provide the system as input along with its format. **Hand written diagrams** should be drawn here showing screen displays and printout formats. You will no doubt change these as the system develops but they are useful to clarify your thoughts. Don't forget you are still developing your ideas here and this will affect the marks awarded for **Clarity of the plan of action**!
- After the output and input have been decided upon you will be in a position to consider the make up of files and record structures. These need to be designed carefully and fully described here.
- **DO NOT give screen dumps [illegible]ur forms, reports nor file structures as technically these haven't been done at this point!**

Here's an example of a form drawn in Word (or Paint?)….



Notes from IB Conference, Bahrain 2000 regarding DESIGN

- Create a detailed representation of the algorithms used (via pseudo-code, structure diagrams, etc) that clearly illustrate your solution.
- Create a list of the variable names you'll use and what each of these variable names will represent
- List the procedure names you'll use and include a description of the purpose of each procedure.
    - The descriptions can later be used as comments

- Use top-down analysis (Break the problem down into smaller problems. These are then broken down in turn until ultimately a pseudo-code representation is obtained which can be used as a basis for program construction.).

    1. Start with a textual description

    2. Then draw a structure diagram such as the **Hierarchical structure** shown below or HIPO chart (which includes input and output) as shown in Bradley text, page 307 (note the textual explanation of each procedure, which is excellent).

PAYROLL PROGRAM structure chart

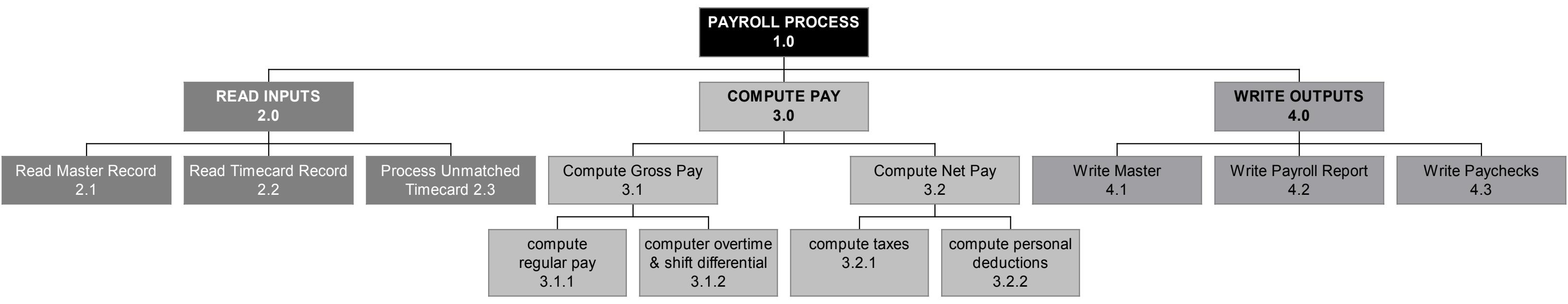


Figure B-1. A structure chart. (the numbers refer to more detailed diagrams of these functions). Each box represents a procedure. The procedure names in the boxes will be the actual procedure names used in the final program.

3. Then you could use Jackson style of Top-Down Design – with procedures (see Bradley, page 305-306: note that the final boxes are in Pseudo-code

4. Finally, It is highly recommended that you place a section of the chart on the left side of the page, with the pseudocode on the right side of the page.

- What you're doing it taking someone who doesn't know programming step-by-step through your program

Notes:

- You don't have to follow PURE *exactly*, but try your best.
- Don't use Flowcharts, only use system flowcharts, top-down design charts.
- If you copy and pasted and use "search and replace" to change the := to back arrows you will reach an achievement level of 1/6.
- When marking this section, the examiner will read the first sentence of the Achievement Levels, and if it is appropriate, that will be your mark (Zero). Otherwise, he/she'll read the second Achievement Level to see if it's appropriate, and if it is, then that will be your mark (1)…and so on… (just like an IF structure).

Do I have to write the Pseudocode BEFORE writing the program?

YES

… "Pseudocode should be written BEFORE writing the program. The candidates might be tempted to write the program first, test and debug it, then produce the pseudo-code AFTERWARD, by copying the program and removing semi-colons and changing "=" to "<--". This is not what is expected, and the students are probably penalizing themselves because this is extra work with no benefit. Some of my students are resistant to writing pseudo-code before writing the program, but all of them agree that it is actually a useful tool (once they get used to it), and they actually finish quicker if they write pseudocode.

Dave Mulkey , IB OCC discussion thread 01/05/2003 09:20:50 PM

Do I have to write the final stage in PURE?

NO, but it is recommended.

"The design process should continue to a detailed stage.
"For example, writing "search for and remove duplicate names from the array" is sensible in the middle stages of design, but the details of the algorithm are supposed to be written before the candidate starts writing program code. This does not mean they must write detailed algorithms in PURE. A statement like "increment counter" is perfectly acceptable instead of "counter <-- counter + 1". Some teachers have students create detailed flow-charts - that's fine, as long as the flow-charts are very detailed. The creation of highly detailed flow charts is pretty time consuming, so I can't recommend this.
"Many schools are having their candidates use PURE to finish the design, as it is a convenient solution. I have my students produce an equivalent level of detail using a formatted English style, e.g. "increment counter".

Dave Mulkey, IB OCC discussion thread 01/05/2003 09:20:50 PM

C. Data Structures [3 marks]

ASSESSMENT GUIDELINES for Using Appropriate Data Structures: [p. 55]

"Data structures that are to be used in the programmed solution to the problem should be **discussed** here. Some sketches/illustration must appear in this discussions, including some sample values."

- Choose data structures that are on the IB HL syllabus
- Choose a Dynamic structure for a Database *Mastery
- Explain in words why you chose the data structures you chose (see Bradley pp498-528 for help. Check other texts and resources as well!)
    - Reading about the data structure you have chosen, will give you more ideas about WHY it's the best choice

ACHIEVEMENT LEVEL for Using Appropriate Data Structures: [p. 49]

Candidates should choose data structures which fully support the data storage requirements of the problem, and which allow clear, efficient algorithms to be written. They need not go beyond the requirements of the syllabus in their quest for the best data structure, nor should they make clumsy choices due to a limited knowledge of the programming language. For example, an array A[1], A[2],… A[10] is usually preferable to the single variables A1, A2, … A10

*3/3: "**All** of the data structures that the candidate uses to solve the problem are appropriate"*

Notes from IB Conference, Bahrain 2000
Include:

- **A discussion** of why you chose to use a **linked list** over an array *Mastery
- Ex. **A discussion** of why you chose a **record structure** over other possible structures such as a one- or two-dimensional array *Mastery
- Ex. **A discussion** of why you chose an array of records over other possible structures
- Ex. **A discussion** of why you chose a parallel array over another possible structure.
- Ex. **A discussion** of why a **list** is more appropriate than a queue or stack or tree
- Ex. **A discussion** of why you chose a **Dynamic** data structure (a linked list or binary tree) over a static data structure.

- **DRAW A DIAGRAM OF EACH STRUCTURE THAT YOU CHOSE WITH SOME SAMPLE DATA IN IT**

Notes from Richard Jones, Singapore 1999

Sometimes data items of different types belong logically together. For example, all of the data about a student in a school could be held in a single object known as a record. Traditional programming languages and applications such as databases provide ways of grouping such data together. The following terminology is associated with records:

The fields are in italic type and the DATA is in CAPITALS. All three records together would make up a data file.

The same information can often be presented in column layout, when all three records can be seen together with the fields used as labels at the top of each column:

(diagram)

Now the file is represented by all of the information in the table. The data in each field can be of any primitive type or of other classes too. Traditional languages provide data structures called **records** which allow data to be kept together as a single unit:

```
newtype  StudentRecord  Record

    Forename     string
    Surname      string
    BirthDate    string
    Sex          character
    Year         integer
    Form         character

 endrecord
```

a collection of these records can be kept in arrays of records:

```
int MAXRECORDS = 50;                     // A constant value, can be changed here
                                                  // and the value MAXRECORDS
                                       used
                                                  //  wherever needed in the Class.

StudentRecord [ ]  recordArray  =  new  StudentRecord[MAXRECORDS];
```

D. Algorithms [3 marks]

ACHIEVEMENT LEVEL for Using Efficient Algorithms: [p. 49]
An efficient algorithm is usually one that executes rapidly and requires minimal storage (makes good use of available memory), but it could also refer to one that uses simple programming code. Even though it is not faster, a loop for printing 20 numbers is more efficient than 20 print commands, because future modifications in the code will be much easier if a loop is used. Such immense inefficiencies in the code can be considered here. It is not necessary for candidates to present quantitative evidence of the efficiency of the algorithms. [but highly recommended!]

3/3: "**All** of the algorithms that the candidate uses to solve the problem are **efficient**."

- **DO RESEARCH_**
  - Find out WHY it's better to use one type of **search (that searches directly)** over others and discuss how your problem is best suited to this *Mastery
  - Find out WHY it's better to use one type of **sort** over others and discuss how your problem is best suited to this *Mastery
  - Find out WHY it's better to use a CASE structure instead of IF (or IF instead of CASE) and discuss why your problem is best suited to this.
  - Find out WHY it's better to use **Direct Access** to records instead of sequential access and discuss how your problem is best suited to this
    - Discuss your algorithms for **adding**, **deleting** and **searching directly** for a record in a file, and prove that you have directly added, deleted and searched. *Mastery *Mastery *Mastery
  - **ETC**
  - Discuss WHY you needed to create a **user-defined function**, and explain HOW it works. *Mastery
  - Discuss WHY you needed to use **recursion** where you did (instead of a regular while..do, repeat..until or for..do loop), and explain HOW it works *Mastery
  - Discuss why the two sorted data structures need to be **merged** and why you chose to them the way you did *Mastery

Notes from the Chief Examiner Glenn Martin:
- The examiner will mostly look at your Pseudo-code for this section
- The examiner is looking for "elegant algorithms"
- You *could* do Big-O analysis
- Be sure to mention other possible methods that you chose ***not*** to use (and why your choice is best)

E. Testing Strategy [3 marks]

ASSESSMENT GUIDELINES for Designing a Testing Strategy: [p.56]

A strategy for testing the program must be included which specifies how the program is expected to behave when differing set of valid and invalid data are inputted. Thus, the strategy should include consideration of what happens when various branches of the program are activated and must consider/discuss the consequences of inputting invalid data. To do this, a table of inputs and expected outputs should be submitted for a wide range of valid and invalid data.

ACHIEVEMENT LEVEL for Designing a Testing Strategy: [p. 50]

*Testing should follow a plan. It should never be random but should be thorough, well organized, and well documented. A comprehensive testing **strategy** is desired; that is better than an immense number of random test cases. Testing plans should be describe and indicate what candidates consider to be typical data and what results are expected. Candidates should also indicate what situations could arise if invalid data is used. This criterion is separate from the hard copy results of the tests.*

3/3: *The candidate outlines a testing strategy that uses a wide range of valid data as well as some invalid data.*

Notes from Chief Examiner Glenn Martin, Bahrain, 2000:

- Ex. “Add Student to a file” will be tested. “Here’s the data I’ll use for this test because it’s standard/typical data: __, ___”. Then: “This is what I expect: It will be added at the first location”.
- Ex. Deleting. Do three test runs. Then add a new data time and make sure that it goes to the deleted spot from previous deletion. IE. ** **Make Sure You Use Continuous Data Testing** **
- Show the Testing Strategy BEFORE you show the Test Output
- Students who achieved low marks on this section just annotated some test runs and printed them. This is bad. Instead: **You must decide what data you’ll try before you do the testing and why.**
- Make a chart. You could use a format something like this:

DATA USED	REASON	EXPECTED RESULT
ADDING A RECORD: Student Name: ___ Class: __ ….	“typical data”	Record is added to file at the first free location
Student Name: __ Class: __ Age: 512 …	“extreme data” (age out of range)	Data rejected
Student Name with a spelling mistake	“typical data”	Data still added

Notes from Richard Jones, Singapore 1999:

Each and every part of your system must be thoroughly tested during development. You should choose standard data to demonstrate normal working. Extreme data should be chosen to show how well your system copes near to extreme limits. Finally abnormal data is used to show that the system will not fall over when the user does something unexpected. The users will do some incredible things from time to time!! You will find this out in your interview!!

This is often a section that is poorly attempted by students. They think that showing their teacher that the system works is good enough. However, your job is to prove to the **examiner** that you know what you are doing. Don't forget that when you choose your test data you should list it and say why you choose it. Then include screen dumps and printouts of the results in the documentation. See Appendices at end.

F. Test Output [3 marks]

ASSESSMENT GUIDELINES FOR HARD COPY OF TESTING: [p. 56]

- The hard copy output should demonstrate the implementation of the testing strategy.
- One or more sample runs should be included to show that the different branches of the program have been tested; testing one set of valid data will **not** be sufficient.
- The hard copy submitted should demonstrate the program's responses to inappropriate or erroneous data, as well as to valid data. Thus the usefulness of the error-handling routines mentioned above should become evident.
- While at least one complete test run must be included in the dossier, it is not necessary that hard copy reflect every key stroke of every test run.
- a pasting of additional test runs can be done to illustrate the testing of different aspects of the program.

- **All test runs should be annotated** in such a way that the candidate is stating what aspect of the program is being tested. Sample output must **never** be altered by hand, computer, erased or covered up.

- Sample output can be 'captured' and combined electronically with explanatory annotations into a single document. However, it is forbidden to alter or reformat sample output in any fashion (except adding page numbers or annotating in order to highlight use friendliness or error-handling facilities as discussed above), especially if these alterations would give an unrealistic impression of the performance of the program. Examples of such 'abuse' include: lining up text which was not originally aligned, adding colour or other special effects, changing incorrect numerical output, erasing evidence of errors.

ACHIEVEMENT LEVEL for Including an Annotated Hard Copy of the Test Output: [p. 50]
The hard copy of test output should demonstrate the implementation of a thorough testing strategy (as indicated by criterion E). It may not be feasible to supply sample output for every single test-case in the testing plan – indeed, a procedure may be designed to automate the testing process and may run many millions of tests. However, at least a representative sample of the test-cases must be presented as sample runs. The output should cover the entire range of test-cases (both valid and invalid data) in the testing plan, and should be presented in an organized fashion (ex. annotated). The teacher must confirm that each candidate has actually completed the testing as claimed in the documentation (see Vade Macum.)

3/3:	*The candidate includes an annotated hard copy of test output based on a wide range of valid data as well as some invalid data.*

Notes from Chief Examiner Glenn Martin, Bahrain, 2000:
Also copy and paste your table from the previous section and add a new column called actual result.

DATA USED	REASON	EXPECTED RESULT	ACTUAL RESULT
ADDING A RECORD: Student Name: ___ Class: __ ….	"typical data"	Record is added to file at the first free location	It works. See test run F1 AND Contents of updated file F2
Student Name: __ Class: __ Age: 512 …	"extreme data" (age out of range)	Data rejected	Data rejected with error message in error trap. See error message H-5
Student Name with a spelling mistake	"typical data"	Data still added	Accepts spelling mistakes for Student Name

G. User-Friendly Features

ASSESSMEMT GUIDELINES for Evidence of User Friendliness: [p. 56]

Evidence of helpful menus, instructions, etc. that will aid the user in navigating through their program must be highlighted. This can be done in the following three ways:

- By annotating relevant parts of the hard copy output
- By annotating parts of the program code that generate menus an instructions
- By reproducing (cut/paste) relevant parts of the hard copy output and/or by reproducing parts of the program code that generate the menus and instructions.

The length of the documentation will depend to a great extent on how much interaction there is between the program and the user.

ACHIEVEMENT LEVEL for Incorporating User Friendly Features [p. 51]

Candidates should give attention to issues of usability during the design stage. The documentation should include some explanations of the reasons for some of the usability decisions. To be given credit candidates must include features which make the program more user-friendly, such as:

- *helpful menus,*
- *help instructions,*
- *useful guidance to the user during the execution of the program.*

These should be documented some way, for example, if an output screen is particularly well designed for readability a hard copy should be provided and labelled as such. Screen dumps and even photographs may be helpful for this criterion.

2/2:	*The candidate incorporates **many** user-friendly features in the program.*

Notes from Chief Examiner Glenn Martin, Bahrain, 2000

The User Friendly Features are FOR THE USER (and Handling Errors)

- User friendly features can be linked to the test runs
- Error messages must be clear
- MUST have:
  Helpful menus
  Help instructions
  Clear error messages
- The user should always be clear as to what to do
- SHOW SCREEN SHOTS to prove.

H. Handling Errors [2 marks]

ASSESSMENT GUIDELINES for Error Handling Facilities:

Error handling facilities can be highlighted in the following two ways:

- by annotating parts of the program code that contain error handling routines
- [better:] Reproducing (cut/paste) parts of the program code that contain error handling routines together with the relevant parts of the hard copy output.

ACHIEVEMENT LEVEL for Handling Errors: [p. 51]
This refers to detecting and rejecting erroneous data input from the user, and preventing common run-time errors caused by calculations and data-file errors. Candidates are not expected to detect or correct intermittent or fatal hardware errors such as paper-out signals from the printer, or damaged disk drives, or to prevent data-loss during a power outage.

The candidate must attempt to trap errors. If no error traps are needed, the candidate must explain why not (example, if all data has been previously checked).

2/2:	*The candidate includes documentation that shows **many** error-handling facilities in the program.*

Notes from Glenn Martin, the Chief Examiner Glenn Martin, Bahrain, 2000

Include Steps to Correct Errors, ESPECIALLY RUN-TIME AND DISK-FILE ERRORS

- Suggestion: Place error traps in Testing with Testing Strategy and make a note of this in this section.
- Candidates must attempt to trap errors! If no error traps are needed for a certain part, you must say why (for example, if all the data has been previously checked by another part of the program).

Make a chart:

ERROR POSSIBILITIES	STEPS TAKEN TO SOLVE

I. Implementation [3 marks]

ACHIEVEMENT LEVEL for Implementing the Program [p. 52]

Evidence here generally refers to hard copy output. Also, the design process should have been sufficiently thorough so that the resulting program has not had to be drastically restructured during the debugging phase.

3/3:	*The candidate **includes evidence** that the program functions well, **AND** the program is CLOSELY related to the design.*

Note from Chief Examiner Glenn Martin Glenn Martin, Bahrain, 2000:

- Include sentences (or a chart!) that say "_____ works because_____", and **summarize** the testing, error handling, etc.

Can I submit a program that doesn't work?

"A dossier is version 1, so applying industry standards, I would expect numerous bugs, as well as quite a few missing features. That is a pretty normal description of an IB Comp Sci Dossier.

"I've seen a number of HL dossiers where the programs were seriously defective. Often, the candidate states this in the conclusion section. Other times, it is obvious from the sample runs or hard-copy of testing. I don't know what fraction of dossiers are in this category - maybe 10%. So that is a fairly normal situation. They don't necessarily receive extremely low marks, but those dossiers are unlikely to receive top marks - probably 10-20 mark range.

"I don't like it if my students submit program a with large defects (e.g. lots of run-time errors, logic errors in complex algorithms producing substantially incorrect results). I tell them to remove that part of the design and remove the defective sections of code, but express their good intentions in the problem analysis and conclusion.

"Each year I see a couple dossiers where the program never ran at all - it contained compile-time errors, and there is no sample output. I find this very sad, and the students only get a couple marks (5 or less). It doesn't seem a sensible situation to me. I'd much rather see an HL dossier with no mastery points, just some loops and if..thens, if that's all the candidate can manage. At least they should solve the problem in some simple way, but the program runs. They can still get up to 25% (up to 9 marks) if all the documentation and such is outstanding. They probably would only get 5 marks or less in reality, as that would be a very weak candidate. But I **think there is an expectation that the candidates submit a program that compiles and runs.** "

OCC discussion thread 09/05/2001 11:17:30 AM

J Programming Style [2 marks]

ASSESSMENT GUIDELINES FOR PROGRAM LISTING: [p. 57]

The program should demonstrate the use of **good programming techniques**. It should include:
An identification header indicating the program name

- Author
- School
- Computer used
- Programming language used
- Date
- Purpose

The program should possess **good internal documentation**, including,

- Constant, type and variable declarations which should have explanatory comments
- Identifiers which should have appropriate names
- Program and subprogram names which should be easy to identify
- Subprograms which must be clearly separated and have comments for their parameters
- Suitable indentation which should be used to illustrate various programming constructs.

The program should demonstrate a knowledgeable use of the various programming concepts (branching, iteration, etc) that are required. Parameters should always be used appropriately, for example to avoid side effect nonlocals should not be referenced within a subprogram.

Program listings must contain all the code written by candidates, and, if a program listing displays code which was automatically generated by the development system, then this code must be clearly identified and distinguishable from that code written the candidate. Only the code designed and written by candidates must be taken into account when applying the assessment criteria.

ACHIEVEMENT LEVEL for Using Good Programming Style: [p. 52]

Good programming style can be demonstrated by program listing which are easily readable, even by a programmer who has never used the program. This includes small and clearly structures modules which have:

- *abundant comments.,*
- *meaningful identifier names, and*
- *a consistent indentation scheme.*

Convoluted, confusing syntax should be avoided whenever possible. Syntax highlighting and line numbering are not required, but are encouraged if they improve readability.

Notes from Chief Examiner Glenn Martin, IB Conference in Bahrain, 2000:

The following MUST be done:

- Small modules
- Clearly structured modules
- Abundant comments that are useful/helpful
- Meaningful variable names
- Consistent indentation that's easy to read

Especially important are:

- Layout
- Meaningful procedure names (which usually start with or contain a verb)
- Asterisks between procedures (even though this isn't done by real programmers)

K. Evaluation [3 marks]

ASSESSMENT GUIDELINE FOR CONCLUSIONS / EVALUATION [p. 58]

This item should include reflections on the effectiveness of the programmed solutions of the original problem. It should discuss answers to the following questions:

- Did it work?
- Did it work for some data sets, but not others?
- Does the program in its current from have any limitations?

A thorough evaluation also discusses possible future enhancements that could be made to the program.

ACHIEVEMENT LEVEL for Evaluating Solutions: [p. 53]

The conclusion should be a critical analysis of the resulting solution.

- ***Efficiency*** *may be discussed in general terms, ex. BigO notation is not required*
- ***Effectiveness*** *should be discussed in relation to the original description of the problem.*
- ***Suggested improvements*** *and possible extensions should be realistic, e.g. suggestions should* ***not*** *include statements such as "the program would be a lot better if it incorporated some artificial intelligence techniques such as speed recognition and natural language parsing".*

3/3:	*The candidate discusses the* ***effectiveness*** *and* ***efficiency*** *of the solution and* ***suggests alternative approaches and improvements***

Notes from the Chief Examiner Glenn Martin, IB Conference in Bahrain, 2000:

- Include three **separate** headings:
  1. "Efficiency"
  2. "Effectiveness"
  3. "Improvements and possible extensions"
- You must **discuss** how **good** it is
- If it's a good quality program then "limitations" mentioned make for much better evaluation!
- Copy your objectives to this section. How you carried out the objectives, the methods you used to carry them out, etc, must be explained/discussed.
- Honesty is expected and works FOR the student, not against.

Notes from Richard Jones, Singapore:

- Perhaps the best way to carry out the evaluation is to look at every single point you made in your objectives [copy your objectives and paste them here] and say if you achieved it successfully. Of course, give reasons.
- All in all, a realistic evaluation of the system you have created is required. You should include the good and bad parts. You must be honest! You won't fool anyone. Say how successful the various parts were. What were the parts that you found difficult to solve. Remember it is the system you are evaluating not the parts of the report. Also there is no need to seek favour by saying that it was a wonderful challenge that you found stimulating. It might have been but that is not evaluation of the system.
- One sentence to avoid is **'I didn't have enough time to complete the project.'** This is often written by the **weaker** students and is likely to discredit your evaluation. There will be enough time - it depends how much you put into the project.

Opportunities For Development

- In the development of your system, you most probably thought of other ways/jobs that would lead to the improvement of it. Explain them clearly.
- Avoid saying "I would like to improve the system by completing all the tasks in the objectives." This should have been covered in the evaluation. This section should be written as if you had completed your system.
- You need to think about new facilities or improved environments, such as:
    - a touch sensitive screen would prevent problems with mouse input,
    - my stock control system could be extended to allow the accounts of the business being kept up to date,
    - use of a relational database would allow greater flexibility in the design of the tables.
- Think about sensible facilities, don't just pick one of the above as it most probably won't fit your system.

L User Documentation [2 marks]

ASSESSMENT GUIDELINES for User Documentation [p. 57]

- User documentation is a simplified set of instructions designed to help end-users operate the program effectively.
- In terms of the hardware, this documentation should provide information concerning the minimal computer system configuration necessary [including memory, speed]
- The documentation should include any information required to **load, start and run** the program.
- A user must be provided with step-by-step instructions of operating the program, being told clearly:
    - What inputs are expected during various stages of the program's execution
    - What outputs can be expected

ACHIEVEMENT LEVEL for Including User Documentation: [p. 53]

Good documentation usually includes both sample output and written instructions. It should be sufficiently complete that it will allow anyone unfamiliar with the program to start using if effectively after reading the instructions. This criterion does not refer to internal instruction.

2/2:	*The candidate includes **clear and thorough** instructions about **loading** and **using** the program.*

Notes from Richard Jones, Singapore 1999

- This is a separate section. You could even make a small extra booklet to show its distinct nature from the rest of the report. In this section you must explain how to run your system to any prospective user. Remember that most users are not knowledgeable about computers. Some are even scared of them. Include here also some sample runs for the user to understand exactly what is happening.
- A common mistake here is to say that the sample runs have been included in the testing section. While it may be true that some of the testing runs will be useful for the user it is important to include them within this section. It is pointless given a user a booklet on how to run your system if most of it refers to something else. Therefore, just copy and paste any useful screen shots.
- Remember screen shots for EVERYTHING!!!!

Due Dates 2003-2004

FIRST SYNOPSIS	Monday September 29th 2003
FINAL SYNOPSIS	Monday October 6th 2003
1. Analysis of the Problem	**Friday October 10th 2003**
--PROJECT WEEK & HALF-TERM HOLIDAY	*October 13th - October 26th 2003*
2. Design: including Charts, Pseudocode, Timelines, Data Structures & Algorithms	**Monday December 1st 2003**
3. Testing Strategy	**Monday December 15th 2003**
--CHRISTMAS HOLIDAY	*December 18th 2003 - January 4th 2004*
COMPLETE most of THE PROGRAM	**January 5th 2004**
--CHINESE NEW YEAR HOLIDAY	*January 19th - 25th 2004*
--MOCK EXAMS	*Wed. January 28th, - Fri. February 6th*
COMPLETE THE ENTIRE PROGRAM	**Monday February 16th 2003**
4. Hard Copy of Test Output + evidence of User-friendly Features + evidence of Error-Handling + evidence that the program works + evidence of good Programming Style	**Monday March 1st 2004**
5. Evaluation & User Manual	**Monday March 8th 2004**

FINAL SUBMISSION DATE FOR ALL PARTS:

MARCH 19TH, 2004