-- Overview --

Your final grade in IB Computer Science comes from 3 individual grades:

- 35 % = **Internal Assessment Dossier** (Programming Project) due March 2007
- 32½ % = **Exam Paper #1** (short problems) May 2007
- 32½ % = **Exam Paper #2** (long problems) May 2007

The Internal Assessment project (dossier) counts a bit more than either of the written exams. It provides an opportunity to develop and demonstrate **programming skills in Java**. The written exams include some programming questions, so the Java skills also pay off in the exam (although the majority of exam questions are not about programming). So it's worthwhile to devote considerable time and effort to the Java programming project.

-- Project Outline --

The Computer Science Internal Assessment project contains a **computer program,** in **Java**, which solves a **real(istic) problem**. But the project requires **more** than just writing a computer program, and is completed in **4 stages**:

- **A = Analysis** - investigating and analysing the problem - [to be done in **October]**
  a detailed **analysis** of the **problem** with a **prototype program,** leading to a **set of GOALS** that will guide the design and be used to **test** the success of the program

- **B = Design** - designing the program (before programming) - [to be done in **November]**
  a pre-programming **design** of a complete **solution** describing **data-structures, algorithms, modules,** and **mastery items** that will be included in the solution

- **C = Programming** - writing the Java program - [to be done in **December and January]**
  demonstrating programming skills **mastery of 10 specific techniques**

- **D = Documentation** - paper documentation about the solution - [to be done in **February]**
  paper documentation showing **thorough testing** of the program, **user instructions,** and **evaluation** of the success of the solution.

The project will be due on **Monday, 12 March, 2007** . It is **all** submitted on paper which will be marked by the teacher after a **30 minute interview**, during which the **teacher and student run the program**. Interviews will be scheduled between **12 March and 26 March**, 2007.

-- Project Requirements -

The student must write a **Java** program. The program must demonstrate that the student has **mastered 10 specific programming techniques**. Some of these are straightforward and others are rather difficult. There is a **VERY LARGE PENALTY** for any missing mastery skills. So it is **essential** to choose a problem and plan a solution that **sensibly** uses the required techniques. Some **problems** are not appropriate (for example video games) and some **solutions** are not appropriate (for example Java applets in web-pages) .

The **official IBO documents** are considerably longer and more detailed than this summary.
Syllabus: http://occ.ibo.org/ibis/documents/dp/gr5/computer_science/d_5_comsc_gui_0605_1_e.pdf
Support: http://occ.ibo.org/ibis/documents/dp/gr5/computer_science/d_5_comsc_tsm_0505_1_e.pdf

-- Paper Documentation --

The project is graded by the teacher, but sample projects are sent away for **moderation**, so the teacher's marks are **not final**. **Paper documentation** is **extremely important** in this project - the teacher will run the program, but the IBO moderator will only receive paper documentation - the moderator **will never run the program**. So it is essential that:

- **thorough records** are kept during the analysis stage (don't throw away your notes)
- a **complete design** is produced **on paper** during the design stage (do this **before** writing the program)
- the program listing is **clear and well structured** and contains **ample comments** (uses good **style)**
- **all** the program's features and functionality are tested and the tests are **recorded on paper** (**LOTS** of testing)

All the documentation and records can be collected electronically, but in the end they must be printed and submitted on paper.

-- Mastery of Programming Techniques --

A key part of the project is **demonstrating mastery.** Each student must clearly demonstrate successful use of a variety of programming techniques. There is some flexibility in the choice of techniques (10 chosen from a list of 15), allowing students to use techniques appropriate to the problem they are solving. But this flexibility is limited, so most students end up choosing a **data-base oriented problem** to make it easy to use the required techniques.

-- Choosing a Topic --

The most important considerations when choosing a topic are:

- the student **understands** the problem (you can't write a chess program if you don't play chess)
- the student **identifies** an intended **end-user** (this must be a real other person)
- there is sufficient **potential** for demonstrating required mastery techniques
- the problem can be adequately solved using the **student's programming skills** and the **available hardware and software**

-- Things to Avoid --

The following are forbidden and/or discouraged:

- **collaboration** - students must do the work alone - especially writing the program - "teamwork" is not permitted
- **animated graphics** - although not actually forbidden, animation cannot be documented on paper so it cannot be rewarded in the assessment criteria. Thus, a video game is not an acceptable topic.
- **copying Java code -** if Java code is copied, this must be **clearly documented** in the program listing, and the student will receive **no credit** for that part of the program. For example, a copied method that sorts an array would **not** be given credit for **mastery** of sorting. If standard solutions are downloaded, they should be used for **clearly separate functions** than the parts the student is programming.
- **writing the program first -** although a **prototype** is a required part of the analysis and design stages, the actual program must be written **after** the analysis and design are done
- **writing for yourself as the only intended user** - the program might be useful to the author, but there must still be **a separate end-user** who is involved in the analysis and testing stages.

Project Topic Areas - The Good, the Bad and the Painful

- **Databases** (straightforward - most students do this)
  Library circulation, Contacts (telephone numbers and addresses), Inventory, Video rentals, Sports, personal calendar, colleges DB, CAS activities DB - many possibilities around school

- **Games** (a bit harder)
  Board-games (tic-tac-toe, chess), Gambling games, educational online quiz.
  Some ideas at: <http://www.mazeworks.com/home.htm>

- **Text-File Processing** (tricky - better for HL)
  Mail-merge, Spell-check, File format conversion (HTML to text), Language translator, Index creator, Web-page builder, extended search/replace, templates, macro text-insertion

- **Simulations** (rather difficult)
  Queuing simulation, Gambling simulation, Physics experiments (pendulum, linear motion and momentum), pond ecology

- **Mathematics** (rather difficult)
  Calculator, Drawing graphs, Geometry, Statistics, Charts, Financial calculations, Mortgages, Calculating prices for pizzas with various toppings

- **Tools/Utilities** (generally quite difficult)
  Compiler/Interpreter, File manager, Compression, Encryption, Graphics file viewer, Automatic process scheduler, Printer control and/or layout

- **Network-based** (tricky and non-standard, disaster-prone)
  FTP client, chat, e-mail, file exchange/storage, web-site searcher, online multi-player games

- **Resource Management** (generally quite difficult - most students don't understand these)
  Train-route planner, PERT/CPM analysis, Scheduling classes, Balance an airplane's freight load, Distribute seats in an airplane

- **Multimedia** (probably **too** difficult and/or inappropriate**)
  music-track editor, animation builder, video editor, video game

- **Real-Time** (probably **too** difficult and/or inappropriate**)
  Robot control, Sensors and switches (temperature sensors and light switches), Traffic lights, Elevators, Burglar alarms

** The last two categories are generally inappropriate as it is **not possible** to provide sample-output on paper.

Finding a Suitable Problem

Candidates must ensure that the problem provides scope for fulfilling the **Mastery Factor** requirements. If you are unsure, **ASK THE TEACHER!**

A **data-base** oriented problem generally provides the most straightforward way to meet the IA requirements and criteria. Data-files are immediately included, and there is an obvious need for sorting and searching. However, many successful dossiers have been based on other types of problems. The use of data-files need not be "central" to the program. If a non-database problem is chosen, teachers should help the students find sensible, appropriate uses for files. A few ideas:

- **Results file** - A simulation or math program can store results in a data-file for later analysis. This is especially sensible if the program creates large quantities of results, such as a simulation producing output every minute for a 24 hour period, or a graphing program generating hundreds of coordinates.

- **Keyword Validation** - Many input operations require a word to be typed which is in a list of "valid" entries - for example, names of months, names of geometric figures, etc. These could be stored in a data-file, along with associated information. In the case of geometric figures, there may be accompanying information such as a definition, formula for the area, etc. The validation routine might permit on-line additions to the keywords, like a spell-check which allows additions to the dictionary.

- **Conversion/Translation** - A program could be "internationalized" by allowing constants and instructions to appear in a variety of languages. Other programs may benefit from conversions of scientific units, money units, etc, where a list of conversions is stored in a file.

- **On-Line Help/Instructions** - Most programs can benefit from some on-line help. This could be stored in a data-file, with help/instruction text stored together with key-words to identify the topic(s).

- **Log-File** - In a tutorial/on-line quiz situation, store results, best times, best scores, etc. Print error-messages and warnings into a log-file instead of on the screen. Store passwords and ID numbers in a file. In each of these, the resulting data-file must be manipulated and used in a meaningful way - e.g. a students' progress in a tutorial system is monitored and the level of difficulty raised or lowered in response to their achievement level.

If files are used in a **supplementary** fashion (non-central), the application should also provide some "utility" features which allow maintenance or analysis of these files - e.g. adding, deleting, or correcting words in a dictionary or validation list, or sorting or searching a log-file before printing. The use of data-files need not be central, but it must be non-trivial - for example, programming a game and then saving the top scores in a data-file is not sufficient to show mastery of data-file operations. Displaying user-instructions stored in a file is also not sufficient for demonstrating mastery of files. On the other hand, a **user-ID-login** feature that looks up the user's name and verifies their password is probably sufficient to show mastery, as long as: (1) the file can be changed in a rational way to add new users or change passwords; (2) the presence of this feature **contributes in a meaningful way** to meeting some of the **goals** for the project. For example, a user log-in would probably not be useful in a mathematics graphing program.

At HL, a similar concept (supplementary use) applies to linked-lists and/or trees. The teachers should feel free to help the students identify suitable, sensible uses for dynamic data structures. The candidates are not required to select problems where the dynamic data structures occupy a central role. Nevertheless, their use must be significant, not a meaningless add-on.

Further Suggestions for Avoiding Difficulties

Sample Data Files - Wherever data-files are used, documentation should contain complete, annotated **listings of the contents** of some sample data-files, so the examiner (and/or teacher) can see the effects of various operations, and see that the operations have functioned correctly. Some sample data-files might be quite large - e.g. hundreds of records or more. These need not be printed in their entirety, but they do make it easier to understand why some file operations are sensible (e.g. why a binary search is better than a sequential search, or why a search operation saves time), as well as providing a more realistic testing environment.

Realistic Problems - Although not strictly required, a "real" or "realistic" problem generally provides more opportunities for the candidates to do some investigation and pre-coding design, and makes the task of writing user instructions more sensible and easier. Realistic problems make many parts of the documentation process easier.

Familiar Topics - Candidates need to understand the problem that they are solving. For this reason, many candidates choose problems that exist in school or in areas with which they have considerable familiarity. Some examples:

- Library circulation and customer data-base
- On-line quizzes/tutorials for school subjects (math, science, English, etc)
- Student grades, attendance, schedules
- School bus routes and rider data-base
- Travel/vacations - exchanging money, dates and times problems, airline reservations
- Retail stores - video lending, inventory, sales, ordering
- Entertainment - TV guide, movies, CD's
- Sports - statistics, planning tournaments
- Parents' business (varies with the student)

-- Interesting Problems (Danger!) --

Many "interesting" problems, such as E-mail, playing chess, video games, graphics displays can be unsuitable for the following reasons:

- The problem is actually too difficult (which is the same reason it is interesting)
- Interesting problems can be very difficult to limit and/or clearly define
- They may be quite difficult to document properly

Interesting problems certainly increase motivation. However, teachers should provide guidance when the students are selecting topics, to ensure that the topic has reasonable scope for complete coverage of the required elements, and that the problem is not too difficult. In many cases it is sufficient to help the student find a reasonable way to **limit the scope** of the solution.

-- Limited Solutions --

High school students **won't succeed** in making **general-purpose commercial software** similar to a professional word-processing program, as they have neither the time nor the necessary skills. It is better to address a **very specific** and **limited** problem. For example, rather than writing a word-processor, the student could create a program that analyses essays by counting words and producing an **index** and a list of **overused** words. Even if this is only usable on text-files (not .doc files), it fills a need that might be missing from the professional word-processor, and is a problem of roughly the correct size and complexity for this project.

== Some Ideas for SL and/or HL ==

Database

The students are asked to write a data-base management system. This has traditionally been a very common choice for portfolios. It naturally stimulates students to cover many of the required mastery topics. Any of the following are appropriate topic areas: phone-book (or contact manager), recipes, dictionary, student courses and grades, teacher grade-book, school bus routes and riders, on-line multiple-choice quiz, inventory, payroll, etc. This is appropriate for SL students.
HL students must be careful to include the less obvious mastery topics (e.g. linked-lists, trees, recursion) - further comments in this regard appear elsewhere.

Web-Page Creator

This can be a template-based, wizard driven web-page creator. It needn't provide full editing features. It is just a quick solution for somebody wanting a straightforward web-page. A good solution would meet the specific needs of the intended user. For example, schools have various events like concerts, presentations, and trips that need to be announced. The intended user might be an activity supervisor like the band director. Such a user will have a pretty clear set of standard information they want included. Simplicity is the goal here - not enormous flexibility.

Digital Camera Album Creator

This might assume the user has stored their photos in a single folder on the hard-disk. Then it takes all those photos and creates an HTML file presenting the photos in a simple layout, linked to an enlarged view when the user clicks. It probably wants to include the ability to add annotations to the photos.

Student and/or Faculty ID Cards

This is a data-base oriented problem which is a bit more exciting than most, as it can involve graphics (photographs) and passwords. (Photographs will only be appropriate if the programming language and the school's hardware support this feature.) It could also involve the students in the graphical layout of the ID card - solving the problem of squeezing the picture and all necessary information onto the card, as well as making it look nice by using interesting fonts and colors. A very good solution might allow the user to customize their own card with their own choice of fonts and colors. Students must not get carried away with the DTP aspects of the problem - they still need to completely demonstrate data-file and pointer skills.

E-mail List Manager

Manage various e-mail lists for various groups. This is especially useful in a school, club, or other large organization which sends regular announcements to various groups of people. This will be a much better solution if some automation is provided, such as actually **sending** e-mails. Good error-handling should prevent accidental mistakes like choosing many groups accidentally. Students should be cautioned that their tests should **not** generate actual spam and annoyance.

Calendar Publisher

Collects events in a database - this part can be quite simple and straightforward. A good solution would then produce outputs in various formats - weekly calendar, monthly calendar, screen or paper versions, alphabetically sorted. The solution will be better if it requires minimal input from users and produces maximum outputs.

== Specific Ideas for HL ==

Stock-Market Prices Analysis

Stock market investors want to analyze historical data (past few weeks or months) as a basis for choosing stocks to buy and sell. Typically the data would be captured from a nicely formatted text file, but that might not permit HL students to demonstrate mastery of file operations. Students are inclined to generate random sample data rather than using real data. However, they should be encouraged to find a source for real data and thus produce a more realistic, usable solution.

Board Games and Puzzles (not video action games)

The knight's tour, 8 queens, and other "traditional" board-game puzzles provide stimulus for trees, stacks, and linked-lists for an exhaustive search for a solution. The student must find an appropriate need for data-files - otherwise they will have difficulty achieving the 100% mastery factor. For example, chess playing programs store an "opening book" in a large data-file. A program which allows users to solve the 8 queens problem might store all the known solutions in a data-file, or collect successful solutions produced by users.

Simulations (Queuing, Physics Experiments, etc)

Students should simulate something with which they are already thoroughly familiar. For example, a billiard-ball simulation should only be attempted by a student with sufficient mathematics and physics knowledge. As with other non-data-base projects, it may be difficult to find a sensible need for data-files here. Students may tend to produce very inflexible solutions using many iterative calculations and techniques. They should be guided to use pointers to produce more flexible, robust solutions.

Mail-Merge

This problem requires 3 distinct modules - creating a form letter, creating a data-base, and then merging the two. A student should not attempt to create all three of these modules - typically a standard word-processor can be used for writing the form letter, so the student need not program a word-processor or text-editor. The data-base management functions are quite straightforward but students will probably want to construct a specific record type (e.g. Name,Phone,Address) rather than allowing a general, flexible, user-modifiable record structure. The merging process presents a good opportunity for using pointers. Teachers may need to guide students in limiting their solutions so that the problem does not become unmanageable. For example, merging into a text-file is sufficient - the student need not program a merge into a proprietary word-processing file format such as MSWord or WordPerfect. The project could be designed to use student and teacher addresses that are already available in a data-file somewhere, but the candidate still needs to program some data-file management features (deleting records, sorting, searching).

Math Calculator/Grapher

This is only appropriate for a student with high ability or high interest in mathematics. Plotting fractals, 2-D and 3-D object rotations and transformations, random walk, line of best fit, etc are problems which are not normally handled by standard calculators and software, and thus motivate the student's efforts. There is considerable scope for using pointers here. However, it is easy to get carried away in the mathematics and graphics and forget about the need to demonstrate mastery of data-file operations. Suggestions for sensible inclusion of data-files appear elsewhere.

Intended End-User

There must be a **specific intended end-user.** This must be a real person (or people) **other than the author**. This requirement causes some discomfort for many IB Computer Science students, so they try to avoid it. Students must overcome their discomfort early and engage in **productive discussions** with the intended user during the analysis and design stages. If done correctly, this makes the programming job easier. Avoid the following:

- Don't say "my program is for **everyone**." That makes the problem very difficult to define, and even more difficult to solve. Choose **one specific user** for discussions - you may wish to think about a group of users, but this should be a small group. For example, "all teachers" is a bad choice, but "several teachers in the math department" is a reasonable choice. "All students" is a bad choice, but "some of the IB Diploma candidates at my school" is a reasonable choice.

- Think about the intended user when **choosing the problem**. Making a "personal calendar" is different for a businessman than an elementary school student. The intended user has a significant effect on the problem definition and analysis, the design, and eventually the choice of the problem.

- Let the user **help** you - talk to them regularly. If you are in the middle of writing the program and trying to decide how a specific interface should look, a brief conversation with the user might reveal that they are happy with a very simple interface, or that they actually don't need that feature at all, thus saving time and energy for the programmer.

- **Take notes** every time you are talking to the user. Otherwise, you must keep everything in your head. More likely, you will simply forget and ignore some of the useful ideas that came from discussions with the user.

- Find a **sophisticated user** - one who knows something about computers and uses them often. Otherwise, they will either have lots of impossible ideas (e.g. "I want to use a microphone to talk to the computer"), or even worse they might have no ideas at all (e.g. "I don't care, anything's okay").

Concentrate on the Problem

During this project you should be **building a solution for a problem** - NOT **inventing a problem to match a solution.** Nevertheless, you must choose a problem that **CAN** be solved by writing a computer program in Java. So you will probably start with an **idea** based on some Java program that you have already written - that doesn't mean you are finished before you start. **Careful analysis** of the **problem** will reveal lots of issues you might not think of right away. In the end the success of your project will be assessed against the **GOALS** you set in the beginning. You should:

- **Investigate** the problem thoroughly and carefully

- **Be creative** when thinking about the goals

- **Discuss** the goals with the user and **reach agreement**

- Make the list of goals **as clear and concise as possible** without sacrificing functionality Concentrate on some clear **benefit** for the user - something that makes their life easier

- **Don't** add **extra (cool) features** if they are not actually needed – this only adds more work without improving the solution (and probably leads to lower marks in the IB assessment)

Creeping Featurism and other Designer Disasters

Many IB Computer Science students encounter significant difficulties in their dossier projects. They start out quite enthusiastic, with good intentions, hoping to create a "killer app". Later they find themselves drowning in complexity. As the deadline approaches (in March) they cut corners and leave things out. In the end the program is not nearly as good as they intended and they receive a poor grade. This has happened to many students in the past. What went wrong?

Poor Goals in Stage A

Too many or over ambitious goals lead to failure. Unclear goals lead to confusion and frustration. A clear list of achievable goals is best. **Clearly** identify a few **important goals** and implement them thoroughly and correctly - don't add lots of "cool features". Just say no to creeping featurism!

Incomplete Design in Stage B

Starting with a vague or incomplete design, students thrash around looking for a sensible direction for their program. In the worst case scenario they create several unsuccessful versions and throw them away. If students put a clear and complete design on paper (in stage B), the teacher can assess this and make suggestions and corrections before the student wastes time and suffers during the programming stage.

Little Attention to Mastery Factors

Solutions **must** use 10 specific mastery factors in a **non-trivial** (meaningful) fashion. Attention to mastery factor coverage needs to start during **Stage B**. If the stage B design does not pay attention to mastery factors, it is quite possible to design a very good **real-world** solution but still get a very bad grade. Attention to mastery factors must continue in stage C, where they must be **successfully used.** Remember - the penalty for missing mastery factors is VERY LARGE! Don't ignore them.

Poor Programming and Testing in Stage C

Students often use inefficient, overly simplistic and inflexible programming techniques. Even students who know how to do things better may rely on ineffective techniques like copying and pasting program code. They compound these errors by doing very little or limited testing. If the program doesn't run, it needs fixing. But if it fails over and over again, this usually indicates poor basic strategies. Students need to be willing to **change** and adopt more productive strategies.

Poor Time Management

If you start too late, or work too slowly, or ignore the need for final documentation, it is easy to end up at the end of March with an unfinished project. Plan ahead, work diligently, don't get sidetracked, and get **help** from the teacher all along throughout the project. In the worst case scenario the program **never runs** - resulting in **zero mastery factors!**

Missing Documentation in Stage D (and other stages)

Many students leave the documentation to do at the very end, after the program is all finished and working. Then they keep working on the program until it is too late and fail to produce required documentation. These missing sections always score 0, and hastily done documentation usually scores low marks. Many students include **far too little sample hard-copy output.** This is especially unfortunate, as it carries penalties in several categories. It is also difficult to understand as it is **quite easy to produce lots and lots of pages.** But you need to do it AFTER the program is finished, so the program needs to be finished and tested AHEAD OF TIME. And it is really, really foolish to do the analysis and design documentation after writing the program!

== Analysis == **1 to 2 weeks (5-10 hours)**

Idea/Problem
State the **problem** and describe some of the details of the problem. Include an outline of **existing systems**, as well as describing some intended improvements over existing systems.

Feasibility Prototype
Write a **brief program** to prove the **feasibility** of a computerized solution. This should be very short, with little attention to the user interface. It should attack a couple of the possible technical obstacles, especially where integration with existing systems might cause problems. For example, if the project involves downloading data-files from the Internet, the feasibility prototype should show more-or-less how it is possible to do this.

Find a User
Find a **user** who is interested in the idea. Show them the feasibility prototype if that is appropriate and useful. Collect questions and scenarios (stories). Help the user to describe scenarios, and encourage him/her to think in more comprehensive and precise terms.

Scenarios
Write down **scenarios** (stories) describing various situations with a variety of results. This should be fairly **comprehensive** (covering many aspects of the problem), but needn't be "complete". This might be done in conjunction with the user, or written first and then discussed with the user.

Discuss Scenarios with User
Discuss written scenarios with the user and add more to produce a **complete** picture of needs and wishes. Expand list of written scenarios.

Mock-up Prototype
Produce **mock-ups** of **user-interfaces**, inputs and outputs, and list of data-storage requirements. This should be fairly complete, but need not be very detailed. User-interfaces shown here are more a base of discussion with the user than and actual commitment to specific appearance.

Goals Meeting
Show mock-up to user. Check that the mock-up is consistent with the scenarios. Sit together and write list of **goals**, **features**, and **limitations**.

Criteria for Success
Write **criteria for success** based on mock-up and goals. Get user approval and supervisor approval for these criteria. This document serves a central role throughout the project.

Supervisor Approval
Obtain supervisor approval before proceeding to the design phase. The supervisor should also ensure that the choice of problem and goals are neither too easy nor too difficult - there must be sufficient scope to achieve 10 (or more) mastery aspects.

Comments: These steps needn't occur in **exactly** this order, but more or less. Partial, incomplete, and revised documents should be preserved for inclusion in the appendix of the final project.

== Design == **2 to 3 weeks (10-15 hours)**

Item Extraction
From scenarios and goals, extract **nouns** and **verbs** to find **data items** and **processes** needed**.**

Tasks Outline --> Program Outline
Write an **outline** of the **tasks** the user will be able to perform. Under each task, list the **modules** needed for each task. Modules include: **algorithms, data-structures, and objects (classes).**

Object Model
Make a list of **objects (classes)** needed. Break down the objects into these members:

User Interface
Events
Actions --> with reference to specific methods and algorithms
Data --> properties and data structures should be specifically mentioned

Name each member and provide a **brief** description of its purpose.

Supervisor Approval - Obtain approval for the initial outline before proceeding further.

Data-Structures
Design **Data-Structures** to accommodate data-storage needs of designed objects. Be sure to include **sample data** and expected storage needs (size), as well as **diagrams** to clarify non-trivial structures such as trees. Clearly explain any ADTs, including the **reason** for using them. New ideas will probably occur and require further algorithms to be added to the **object model** (above).

Algorithm Details
Design algorithms including clear, detailed explanations of **how** algorithms will function. This may include pseudo-code and java code snippets to clearly explain the **intended** programming. Complete method headers must be written, but pseudo-code needn't be as precise as Java code. Standard algorithms may be identified by name – e.g. “execute sequential search”. Non-standard algorithms must be described in greater detail. Java test-code may be written to test feasibility, but explanations should **not** be presented as finished Java code.

Refinement
During design, new ideas for **criteria for success** may occur - e.g. reorganizing files, use of other hardware, etc. These can be integrated into the goals document, as long as important user-oriented goals are not sacrificed.

Mastery Factors
Your design must include clear reference to the mastery factors that will be demonstrated. For example, if one of the data-structures is a RandomAccessFile, and there are methods for saving and searching, then that covers 2 mastery items. Make a list of the mastery factors you will demonstrate, with the names of data-structures and methods which will demonstrate them.

Supervisor Approval
Obtain supervisor approval before proceeding to implementation (programming). The supervisor should ensure there is sufficient scope to achieve 10 (or more) mastery aspects.

Comments: Object, data-structure, and algorithm designs must adequately support the goals stated in the **criteria for success**, as well as demonstrating **mastery factors**. The plan must be achievable with the student's programming abilities and available equipment.

== Program Construction == **6 to 8 weeks (30-40 hr)**

Alpha Version
Start with a simple version, with a simple interface and meeting only a few goals - this might be a small extension of the prototype. Develop and debug this version. Accomplish some of the more difficult technical tasks - e.g. build the base of library functions necessary to make the rest of the program work - but it is **not** necessary to achieve **all** functionality at this point. This is a **technical** release, not a user release. **DON'T** leave bugs in this version for later. The bugs are easier to find in this small version than hunting them down later in a big program. The programmer should assess this version themselves.

Beta Version
Add **all** features. Don't worry about usability and error-handling yet, except where these are essential for making this version work. Make sure ALL the mastery aspects are accomplished in this version, and that virtually all the **functional** criteria for success are met. If some of the success criteria involve usability or performance, some of these may be left for the next version.

User Beta Testing
Get the user to spend some time testing the beta version, and to make comments about needed improvements. Most important are complaints about failure to meet **success criteria**. User(s) may suggest usability and performance improvements - note these for inclusion in the next version.

Supervisor Check
Have the supervisor check the **code** to ensure that **mastery aspects** have been met. If not, some significant redesign and reprogramming may be necessary. The code should be thoroughly documented and easily readable at this point. The supervisor should **suggest improvements**.

Finished Version
The finished version of the program should correctly address all the following:

- adequately meet all the **criteria for success**
- demonstrate 10 (or more) mastery aspects
- adequate usability
- adequate error-handling
- adequate performance (speed and/or data storage efficiency)

Usability and Error Handling Documentation
Write descriptions and explanations of **usability** considerations and **error handling** features. Where appropriate this should refer to **criteria for success.**

Incremental Development Cycles - Code, Test, Fix
Development should be cyclic. The Beta version is an **expansion/elaboration** of the alpha version. Each cycle includes programming, testing, and debugging. Comments in the source code and collection of test-cases (and possibly test-harness methods) should document the development.

Debugging
Testing must be documented. Long debugging sessions should produce collection of test-cases that will be presented later. A **journal** may be useful in this context - i.e. “Today I spent an hour debugging the sorting algorithm for the RandomAccessFile, which kept destroying records.”

Supervisor Check - Check that **suggested improvements** (above) were made, and that mastery aspects have been achieved.

== Testing and Documentation == **2 to 3 weeks (10-15 hr)**

User Documentation
Before doing the final testing and evaluation, writing the user documentation will help the programmer organize their thoughts about the program. It may also be useful to write some of this **before** finishing the program.

FINAL Test Output
All the testing in this section must be done **after** the final program has been written. This is **not** a debugging session, but a documentation session. There will probably be failures in the program - these must be documented rather than being fixed. If major problems surface, it may be necessary to go back and **re-**finish the final version. But after that, **all** the testing must be performed again.

Complete Sample Run
If possible, produce a **single session** showing typical use of all the required features, and capture and annotate output for this entire session. This will include only **single examples** of **normal data**, not strange error-provoking situations nor multiple examples of the same feature.

Targeting Criteria for Success
A set of sample output should document successfully meeting the **criteria for success** (A2). This must be comprehensive. It must include **ample** normal data to show the **completeness** of the solution (e.g. lists of data rather than just single data items), as well as **abnormal data** to test the **robustness** of the solution (e.g. proper responses to error conditions.) It must be possible for the teacher to perform these tests, or to sit with the student while they do so.

Usability and Error-handling
Some of the sample output (above) will demonstrate usability and error-handling. The annotations should reflect what is demonstrated in each case.

Evaluating Solutions
See notes in IB assessment criteria. Be sure to address the **criteria for success**, and make suggestions of how a future version could **expand** these criteria.

== **Final Interview ==**
A 30-60 minute interview with the teacher, **after** the teacher has seen the documentation, helps the teacher with the assessment. The teacher should award a **holistic** mark and sign the cover sheet.

Comments: These stages appear to represent a straight-line process, but we know that bugs, mistakes, bad decisions and uncertainty cause programmers to go back to previous stages. "Spiraling back" is inevitable, but students should do this **consciously**, rather than working in an unstructured and undirected fashion. Students should always have a **clear sub-goal** in mind when they are working. Are they testing? Developing? Designing? After the analysis and design phases are finished, students should be implementing the program to **meet** the goals. They must resist the temptation to add more and more features as the program becomes larger.

As the program becomes large and complex, and bugs are difficult to find, adding **required features** becomes difficult - so instead of doing required work, students may create a new goal that is **easy** to implement. All too often new features are actually unnecessary and disconnected. Clever gadgets and cool interfaces (disappearing buttons, graphical decorations, etc) probably contribute nothing to the original goals. If done sensibly, spiraling back to change the original goals would result in a significant rethinking of the design. "Starting over" is the extreme case of spiraling back, and it demands a complete redesign - that should be avoidable by careful decision making during the analysis and design process. Starting over is usually not a realistic option.