IB Computer Science

Internal Assessment Dossier Guidance

Dave Mulkey - Usingen, Germany - June 2009

Preface

The **dossier** is the Internal Assessment assignment for IB Computer Science. The course covers Java programming in considerable detail (more for Higher Level than Standard Level). The dossier is the primary assessment tool for the programming part of the course. It counts 35% of the final IB grade, with two exams counting 32.5% each.

The dossier requires the students to use reliable software design techniques, in addition to using correct Java programming techniques. Since students have "unlimited" time to devote, it should be possible to score a high mark on this project. Unfortunately, dossier marks don't tend to be any higher than written exams. This seems to result from some misunderstandings of the requirements of the dossier. Many students fail to address required elements and lose marks for rather silly reasons.

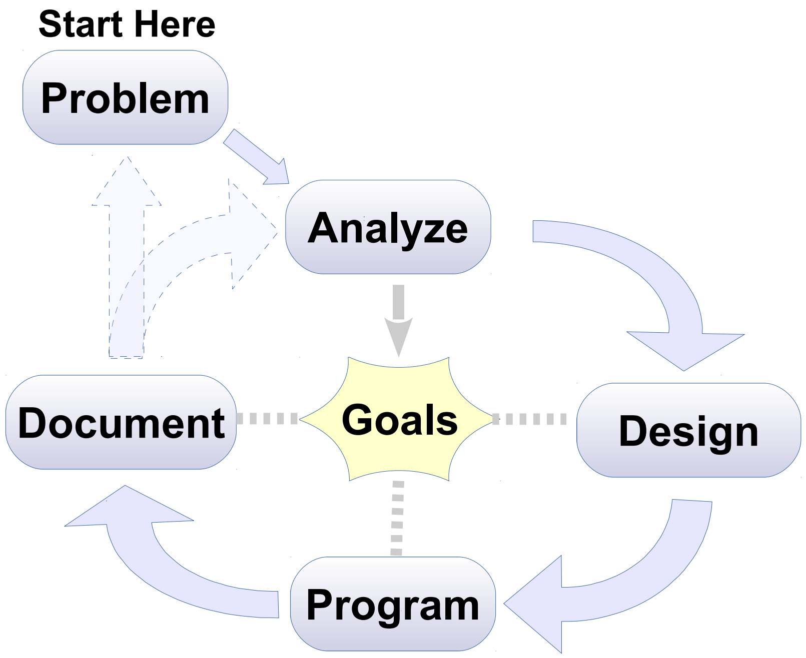
The intention of this document is to help IB students understand the dossier requirements and avoid mistakes, as well as providing suggestions to make the project easier and make the students more successful with less frustration and effort. The recommendations and suggestions are the result of many years of IB teaching experience, plus years of work as an IB moderator for the dossier – but these notes are not "official" in any sense. For "official" answers, refer directly to the IBO Guide and TSM at:

IBOGuide(Syllabus):http://xmltwo.ibo.org/publications/DP/Group5/d_5_comsc_gui_1201_1/html/67.207.142.65/exist/rest/app/gui.xql@doc=d_5_comsc_gui_1201_1_e&part=2&chapter=3.html

Teacher.Support.Material(Internal.Assessment): http://ibpublishing.ibo.org/live-exist/rest/app/tsm.xql?doc=d_4_comsc_tsm_1201_1_e&part=4&chapter=7

Table of Contents

IB Computer Science Dossier Overview



The Dossier

The **dossier** is the Internal Assessment component of IB Computer Science (HL and SL). This is much more than a simple Java program. Students must solve a **real problem**, for a **specific end user**, by **analyzing** the problem, **designing** a solution, writing a **program** in Java**,** and **documenting** the program and resulting system. At the end, students **print out** the program listing and all the documentation and submit it on paper. It is assessed by the teacher (and then checked by IB examiners.)

The diagram outlines the **Software Life Cycle**, **emphasizing** the importance of meeting **goals**, and **de-emphasizing** programming. Writing the program is necessary, but not sufficient.

Real software gets redesigned and reconstructed (software -life-cycle), but the IB dossier only requires going around the big loop one time.

Four Stages

The dossier has **four stages** .

- **Stage A : Analysis**
- **Stage B : Detailed Design**
- **Stage C : The Program**
- **Stage D : Documentation**

The four stages are normally completed over a fairly long period of time – several months. This extended time-line is not a requirement, but most students find it impossible to do it all in a single month.

The IBO Guide recommends *"25 (SL) or 35 (HL) hours of teacher contact time plus further computer access time."[Guide p 48]*[1] - so probably more than 50 hours in total. The actual time commitment is difficult to measure, as dossier work usually overlaps with instruction about Java techniques. And the suggestion of "*60-100 (printed) pages"[Guide p 54]* is realistic.

Before Starting

The dossier requires students to think ahead during analysis and design. Thinking ahead is difficult and requires guidance and instruction. Students must know enough Java to know what is possible (especially as they must design data-structures and algorithms during Stage B), as well as knowing about some reliable systematic methods to use during Stage A.

Many teachers try to start the dossier early in the second year of the course. Before that point, the students don't really know enough. Starting later is possible. However, many students fail to finish on time because they start too late, or because a poor design caused them to waste a lot of time, throwing away work and starting over too often.

"*Preferably students will complete these stages in the order given.*"*[Guide p 53]* However, students commonly encounter unexpected difficulties while programming and must revise the original design before continuing.

A - Analyzing

Stage A starts with **choosing a problem** and **finding an intended user.** Many students want to start by writing a program – but that is not the intention of the dossier assignment. Starting with programming usually leads to a poor design, missed deadlines and general difficulties.

Stage A : Analysis

- select a **problem** AND an **intended end-user**
- **analyze** the problem, using a **systematic method**
- **create a prototype** and discuss it with the user
- set **goals** (criteria for success)

The problem needs to be specific and the goals should address the needs of the intended end-user. Starting with a solution – e.g. "I'll write a game" - rarely addresses a specific problem or a specific user. The Guide says: *"The dossier must address a **single problem** that can be solved using computer systems and which has an **identified end-user.**"* [Guide p. 52]

B - Designing

After a thorough analysis produces appropriate goals, a thorough and careful **design** should make the programming task relatively straightforward, avoiding wasted time and effort when programming.

Stage B : Detailed Design

- **data-structures** (files, arrays, Abstract-Data-Types)
- **algorithms** (methods and pseudo-code)
- **modular organization** (classes = data-structures + methods)

The Guide notes: *"... with stages B and C it may occasionally be necessary for students to return from C to B one or more times to refine their detailed design in a "spiral" of design and development." [Guide p. 53]* So, although students should devote significant time and effort to creating a good design, they needn't worry about making it "perfect".

C - Programming

Students are expected to write a substantial program, *"500-3000 lines" [Guide p.54]* (this is only a suggestion, not a requirement). The program should fulfill the **Criteria for Success** (goals) written in Stage A. That means the program must actually function and it should be possible for the intended user to use the program successfully.

Stage C : The Program

- **write a program** (following the design)
- satisfy the **Criteria for Success** (goals from Stage A)
- make it **usable** (including user-friendly features)
- make it **robust** (reliable) by handling errors
- fulfill the **Mastery Factor** requirement (at least 10 items)

The program must demonstrate **mastery** of Java programming techniques required for the course. That sounds like a "text-book exercise", but it's more than that. Students must write methods and classes that function correctly when the **entire program** runs, not only under simplified test conditions. Students should avoid simplistic solutions. They should use more powerful and flexible techniques whenever possible, as that leads more easily to a successful program and satisfies mastery factors.

D - Documenting

Stage D : Documentation

- **hard-copy** (printed) output of running and testing the program
- **evaluation** – describing what worked well and what didn't
- **user-documentation –** a user's manual

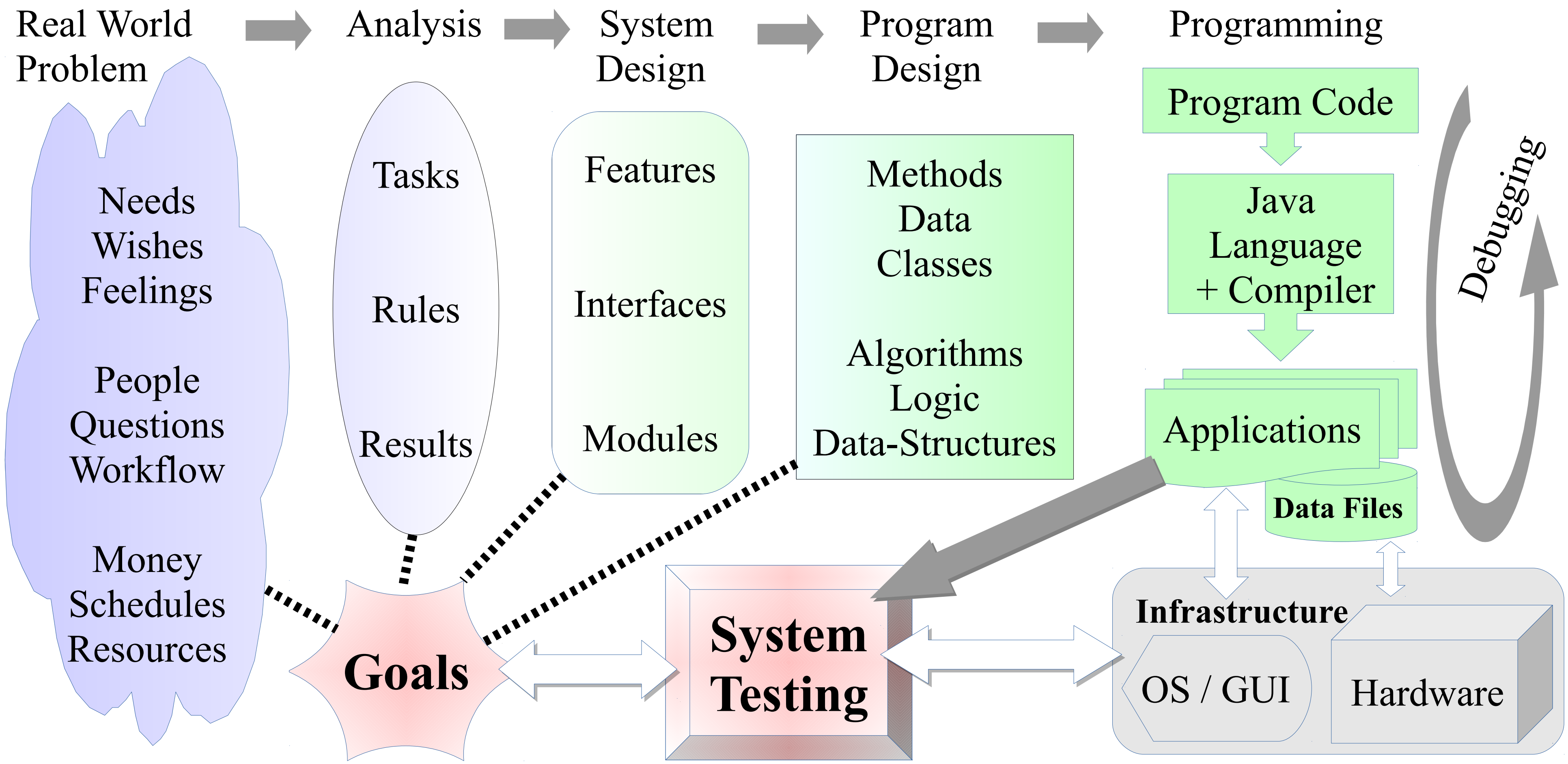
Hard-copy output consists of **screen-captures** of the running program. An IB moderator will never see the program running – they will only see this printed output, so it must be complete and extensive. User-documentation will contain similar screen shots with explanations of how to use the program. The evaluation includes suggestions for improvement.

Planning Before Doing

The dossier requires students to work through a logical, thorough, reliable design process before writing a Java program. It requires students to start at the user's end, analyze a problem to produce goals, work through a sensible design, follow good software development principles, and finish with thorough testing and documentation. Students must work in each of these stages, using appropriate tools and strategies at each stage.

Each stage requires a different **view** of the system. For example, when the user describes the problem, they will talk about completely different issues than a programmer who is trying to produce a solution.

The diagram below outlines different issues at various stages in the **software development** process. It may be overwhelming, but it clearly shows how developers "change gears" from one stage to the next.



Stage A – Analyzing the Problem

The chart above becomes clearer with a brief look at a specific example. (This example is only a very brief overview – NOT a sample dossier.)

→ ***The School Attendance Tracker***[*]

Students have learned about files and arrays and methods before starting on the dossier. So their first thoughts are probably along the following lines:

- We can start with an **array** of all the student names
- The teacher must *choose* the names of absent students
- When finished, a **method** runs to save the absent names
- The absent names must be stored in a **file**

Let's stop now, because this is the **wrong way to start**. We have started immediately thinking about the **solution** – the **program.** You will certainly have this sort of thing in the "back of your mind", but it should not be your focus at first.

Start by focusing on the **problem**, especially as it relates to the **user.**

What's the problem? And who cares?

The problem isn't "taking" attendance, or "using an attendance program", but rather **managing** attendance. Students miss school for various reasons – illness, field trips, dental appointments, meetings with counselors, etc. Then someone (e.g. the teacher) needs to record the absence, someone else records the reason (e.g. a clerk) - schools want to know **why** a student was absent. So there are **two** intended users – teacher and clerk. And a vice-principal might need a list of the students with too many absences – another **user!**

The **user** is an integral part of the **development** process - their **needs** should guide the design of the software.

* This is a **simple** example - students should NOT actually choose this problem.

Use the User(s)

Stages A1 and A3 require students to **discuss the problem with the intended user.** Many students find this uncomfortable and don't know what to talk about. Many users don't want to spend time on this either. All too often the user and the student don't understand each other and don't exchange useful information.

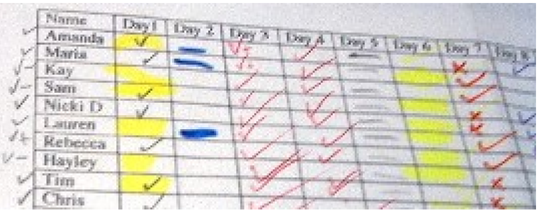
As a student, you probably are not an expert in the field of your chosen problem. Although you attend school, you probably never thought much about the school's attendance system. The user has a lot more understanding of the problem than a student, so students should take advantage of the user's expertise. Otherwise the resulting program can be far too simplistic to satisfy the user's goals, or can become unnecessarily complex and thus very difficult to complete.

If there are lots of users (say 50 teachers), you might want to distribute a **questionnaire** rather than having conversations with so many people. But it is still useful to talk to a specific user. Choose one who is particularly helpful. Let them help you figure out what is important and what is not. Let them explain issues in the **problem domain** that you don't understand. Let them make decisions for you – or let them help you make decisions.

Sample Data

Students are required to *"show evidence that relevant information has been collected."[Guide p 55]* Questionnaires certainly satisfy this requirement. A simpler way is to collect sample data as it exists in the current solution – often on paper. If teachers are doing their attendance in a paper grade-book, make a photocopy and include it in the dossier:

Teacher's Paper Attendance Record



More Sample Data

A single photocopy probably doesn't contain enough sample data to enable the student to analyze the problem fully. Most likely the school already has an attendance system, whether paper-based or electronic. They probably print some kind of reports. Get copies of some sample reports, discuss them with the user, and include them in your dossier. Looking at actual reports and actual data helps focus your thinking.

Attendance Summary (Entire Year)

Student IC/LOB	Attendance Days	Absent Days	Total Days
890410115589	12	9	21
890823115763	6	15	21
890521115493	17	4	21
890120045068	17	4	21
891005115141	20	1	21
890219115513	17	4	21
890122016447	20	1	21
890210115605	10	11	21
890914115857	17	4	21

Daily Attendance Report

Gilbert Cabot		
ampbell Casey		
Davis Copperfield (Dani)	Absent	
ily Crewe		
Donald Duncan		
arlotte Eyre	Late	10:05

Students should also collect other kinds of information, like:
- **rules** (teachers must take attendance in the first 5 minutes of class)
- **difficulties** (paper grade books cannot be shared with the office)

This type of information can appear as quotations or summaries of discussions with the user. These can be recorded as **user-stories.**

A Systematic Method

For maximum marks in section **A1 – Analyzing the Problem**, the student must provide "*evidence that a **systematic method** has been used in the analysis of the problem."[Guide p 55]*

*"A **systematic method** is one that takes into account what input and output will occur and what calculations and processes will be necessary to obtain the desired output."[Guide p 55]*

A simple **systematic method** that emphasizes user-discussions and users' needs is collecting **scenarios** – or **user-stories** (XP)[2] or **use-cases** (UML)[3] The student must take notes during user-interviews and then extract stories about things that happen and tasks the user performs. The student should highlight items in the stories that involve **input, output, processing,** and **storage.** Here are some user-stories about the attendance system.

Taking Attendance ***User Story***
At the beginning of each class, the teachers take attendance. They mark down the names of any absent students. If a student arrives late, they change the attendance to LATE instead of ABSENT. This is messy when done on paper – requires using pencil and an eraser.
Input – name of absent student, period of class meeting, name of class
Processing – At the end of the day, names of absent students must be sent to the office. This is currently done on a standard form.
Output – list of absences sent to office

Excuses ***User Story***
When students return from absence, they must bring a signed note to the office. The attendance clerk then writes "excused" next to the student's name in their book of absences. If a student forgets the note, the clerk calls the parents at home. If the parent says the student was not sick, the clerk records "unexcused".
Input – note or message from parents
Processing – decide whether the absence is excused or not
Storage - write "excused" or "unexcused" in the paper register

Weekly Summary ***User Story***

In the current paper system, the attendance clerk has a paper list of all the student names. Each time a student is absent, the clerk places a mark next to the student's name – either X for excused, or U for unexcused. At the end of each week, the clerk reviews the entire list and writes a list of students who have been absent too often – this is a different number each week. The list goes to the vice-principal who either talks to the student, calls the parents, or both.

Input – list of student names with absences X and U
Process – find students with too many absences
Output – list of students with too many absences

Catching Skipping Students ***User Story***

This does not work very well with the paper system. The office only gets the lists from teachers at the end of the day. So if a student skips periods 3 and 4, it's only noticed at the end of the day. It is also difficult for the clerk, as she must check multiple lists to find out whether a student was absent from one class but not from others. She deals with this by making check-marks on a scrap copy of all the student names, but it is time consuming and unreliable. She wishes this were easier – e.g. automatic.

Input – absence lists from all teachers
Output – checkmarks on scrap paper
Processing – search for students with only 1 or 2 marks
Storage – scrap paper copies are stored in a notebook, but rarely used

This set of stories is **incomplete**. It does not indicate how the clerk knows which students from yesterday need excuse notes. This might indicate an actual hole in the system, or it might just be incomplete information from the user. The student needs to search for missing information and go back to the user to collect any information or data that are missing. Through repeated discussions the student develops a thorough understanding of the problem.

Choosing Goals

Criteria A2 is about **Criteria For Success –** also called **objectives,** or more simply **goals.**

Some goals seem obvious:

- the computer must store the attendance record for each student

But even obvious goals sometimes need clarification:

- the computer must store an attendance record for each student during each period of each school day – so 6 records per day

Other goals are not so obvious. They can be suggested or clarified during user discussions:

- the attendance clerk needs access to ALL teachers' data - so she can follow up and collect excuses the next day
- the vice-principal needs a weekly report of students who have been absent very often – so he can chase down chronic offenders
- teachers do not wish to type in student names, but rather should be able to select them from a list by clicking - so taking attendance is faster, and names are not misspelled

*"The student relates all of the objectives of the solution to the analysis of the problem, and **outlines** the limits under which the solution will operate" [Guide p 56]*

This implies that there are **reasons** for the goals. The reasons (- so ...) needn't be long and complex, but should connect the goals to user stories.

Although not specifically required, most dossiers will attempt to make an **improvement** over the previous system. These improvements can arise from a "wish" story – e.g. the user says "I wish that it worked like this..." and then describes something they want (like the Skipping Students story). Programmers should not invent "cool features" without discussing them with the user to see whether they are actually needed. All the goals should be connected to user-stories. Remember – keep it simple!

Keep It Simple

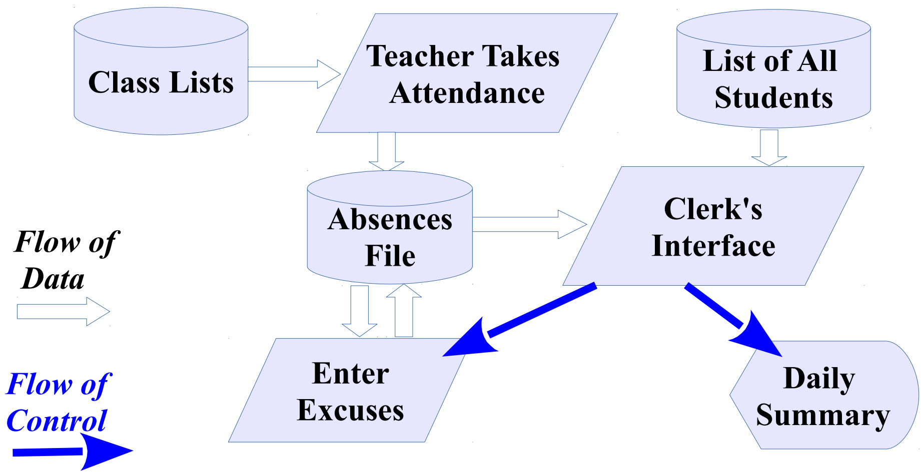
"Creeping-featurism" refers to the temptation to add more and more features to make the program better and better. Unfortunately this usually leads to **unfinished** programs that do **absolutely nothing** – that's NOT better! In the end, all the features in the program must actually WORK.

Initial Design

At some point, the student needs to start thinking about a possible structure for the solution – an **initial system design**. The student should examine the user stories and preliminary goals and think about:

- What **tasks** must be performed? What are some **automated processes** that will support the user in these tasks?
- What **data** is required for the tasks and how should it be input?
- What **results** must be produced? How should they be displayed?
- What **data-files** are needed for storing data permanently?

The initial design can be displayed as a diagram – very simple at first:



So what about the PROGRAM?

Starting with programming, without careful planning and design, is called "code-and-fix programming". *"The second reason code-and-fix development is appealing is that it requires no training ... The sparkle and luster of quick, early progress is one reason that customers and managers continue to fall for code-and-fix development."[McConnell p 15-16]*[4] McConnell points out that careful planning is not wasted time – it actually **shortens** total development time.

Computer Science students are anxious to get down to the business of writing a program. In all honesty, the users are probably also anxious to "see something" - they'd really like you to open up a box of software, install it, and *voila!* the problem is solved.

At this point you're still not ready to write the program, but you can satisfy the urge to "see something" by creating a **prototype.** A prototype is a very simple, preliminary version of the program. By designing a few **user-interfaces**, you can help clarify your own thinking, as well as using the prototype to **show the user** what you are thinking and facilitate discussion.

The Prototype

The next page shows a prototype for the attendance problem. Certainly other prototypes are possible – this is just one example. Also, it could be considerably longer and more detailed, but it's kept simple here for brevity. Students are encouraged to produce these sample screens with a word-processor or presentation tool (if they prefer pen and paper, that's acceptable.) Then discuss the prototype with the intended user(s).

*"The prototype solution **must** be preceded by an initial design for some of the main objectives..."[Guide p 56]* So the diagram above (at left) must be created **before** the prototype. Also, the prototype should **correspond** to the initial design (assessment A3). So DO THE DESIGN before the prototype! (If you prefer, your design could be a text outline or a UML diagram.)

Prototype

Teacher Taking Attendance Date : 01 Apr 2008
Teacher : Einstein, A

Classes	**Students**	**Absent**
Math 9a	Adams, Alice	Baker, Bobby
Math 10a	Baker, Bobby	Eagle, Eddie
Math 10b	Cho, Kim	
Math 11	Duck, Don	
	Eagle, Eddie	
	......	

SAVE

The teacher clicks on a class → the list of students appears.

The teacher clicks on each absent student → each name is copied into the list of absent students.

If a student arrives late, the teacher clicks on the name in the ***absent*** *list and the name disappears.*

When finished, click [Save] .

Data File

```
.....
Peters, Paul / English 9c / Elliot, R / 31 May 2009
Peters, Paul / Math 9c / Gauss, M / 31 May 2009
Adams, Alice / Math 10a / Einstein, A / 01 Apr 2009
Eagle, Eddie / Math 10a / Einstein, A / 01 Apr 2009
Adams, Alice / History 10a / Churchill, W / 01 Apr 2009
......
```

Exact details of this data structure are not necessary – this is just a basic concept so we can discuss ***what*** *data needs to be stored, not how.*

Daily Attendance Summary 01 Apr 2009

Change Date

Print

```
Adams, Alice / Math 10a / Einstein, A
Eagle, Eddie / Math 10a / Einstein, A
Adams, Alice / History 10a / Churchill, W
....
```

Enter Excuses Mercury, Freddie

```
Math 10a    / Einstein, A  / 1 Sep 2008 / Sick Excused
Math 10a    / Einstein, A  / 2 Sep 2008 / Skipped Unexc.
History 10a / Churchill, W / 5 Sep 2008 / ???
  ....
```

Click on any absence to enter (change) the excuse.

Type An Excuse For

```
Mercury, Freddie / History 10a / Churchill, W / 5 Sep 2008
```

[???] → [Sick Excused]

Clerk's Interface

See All Data for Today

Student Name: Mercury, Freddie

Enter Excuses

The Student Name box can be a drop-down list of all student names.

Happy User = Automation

"User-friendly" has been misunderstood as *cheerful, pretty* software that makes users *happy*. The actual key to user satisfaction is **reliable** and **usable** software. This is software that does what the user **needs**, without causing problems or increasing the user's workload.

The prototype above demonstrates the essential features as requested by the users. Now the user might have further questions or suggestions :

- Do I need to type text for the excuses, or can I have a drop-down list?
- How will the computer know the names of the students in my classes? Do I need to type them in, or is it "automatic"?
- The vice-principal would really like automatic notification when a student has missed 10 days of school. Could we add a button on the clerk's page that displays the students with too many absences?
- Can we have pictures of students, so the teachers can take attendance more easily in the first week of school?

Users often ask for **automation**. They don't want to spend all day clicking buttons, waiting for processing and then reading the results. They want as much automation as possible.

Happy Programmer = Reasonable Expectations

After showing the prototype to the end user(s) (e.g. the clerk and some teachers), the programmer might decide to add another file (or files) to the program, containing the entire **schedule** of all the students and all the classes in the school – this is needed to automate the teachers' page. Once this data is added, the programmer might be tempted to add a scheduling module to the program, to make it even more useful. And pictures of the students ... and a web interface ... and some small games to make the users "happy" .... and ... and ... and .... ☹

💣💣 Beware – Creeping Featurism 💣💣

Keeping it Under Control

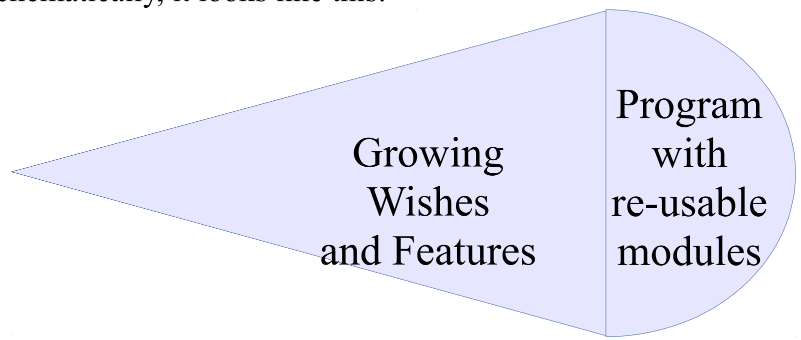
The user's needs and wishes for automation lead to an expansion, and usually **complication**, of the solution. The programmer's needs to solve the problem and meet a deadline drive the solution in the direction of **simplification**.

Young, inexperienced programmers tend to **add more code** to for each feature added to the solution. This generally leads to unfinished software (didn't meet the deadline) or unreliable software (met the deadline but didn't really meet the goals).

In a successful dossier, the student can keep the drive for expansion under control in two ways:

- choose goals and features carefully, providing **only what is needed**
- good programming techniques that **simplify** the program but simultaneously increase it's flexibility and power

Schematically, it looks like this:



By using **reliable programming techniques** to create **re-usable classes**, **methods** and **data-structures**, the programmer can create a **flexible** system. Then the **features** and **usability** of the program can **grow faster** than the size of the program. There is less to write, less to debug, and a bigger chance of success.

This may sound magical, but it only requires careful **design.** That's the reason for **Stage B – Detailed Design.**

Choosing a Problem and a User

What is a "bad choice" for a problem?

The attendance system is a bad idea for an IB dossier because:

- it involves lots of users and that makes everything more difficult
- the school's administration will probably want it to "integrate" with some other software, like the scheduling software – that's difficult
- the other software probably has requirements and security restrictions that are beyond the student's control and expertise
- it probably doesn't really need a solution anyway

Students should avoid problems that require exotic tools and techniques, like:

- web communications
- real-time systems (controlling a robot)
- unusual hardware, like sensors and wireless communication
- video animation (because it cannot be represented on paper)

Systems that the student cannot control cause problems:

- a PC is easier to control than a server
- data-files are easier to control than large data-base systems
- the school is a better environment than real businesses having legal or ethical restrictions (like a doctor's office or a factory)

There are no "bonus points" for choosing an unusual topic, so **Keeping It Simple** is the advantageous - but do try to choose something interesting.

What is a bad choice for a user?

Whichever user works with your chosen problem is a good one, but if possible choose an experienced, competent user – they will be more helpful. And choosing "yourself" as the user is a bad idea (forbidden??)

Suggested Topics

What is a "good" topic area?

Many students have written successful dossiers in many different topic areas over the years.

Data-Base Oriented Problems

The most common choice is a personal database, storing something like phone numbers or web-addresses. This has an obvious need for file-handling, so it's easy to fulfill the HL and SL file mastery factors. It also corresponds well with syllabus topics. Contrast this with communication-oriented problems, that require skills and techniques that are not in the syllabus.

A common difficulty in Data-Base dossiers is a lack of data. Student programmers are reluctant to actually type in lots of phone numbers, so they end up with a program that is difficult to test – they test it only on a few data entries.

Another common difficulty as lack of excitement – Data-Base dossiers can easily become boring and the student loses enthusiasm.

Still, a Data-Base dossier is an easy choice, there are lots of possible topic areas (CD collections, video collections, e-mail addresses, etc) and the solution is usually straightforward.

Something more exciting?

It's pretty easy to dream up "exciting" problems, like making a web-search-engine, or a chat server, or a 3-D ego shooter game. But if it's really exciting, it's quite likely to be too difficult – perhaps not too difficult for the student's skills, but too difficult in terms of time requirements.

Games sound exciting. However, they tend to be much more difficult than first appearances indicate. And anything with animation is discouraged because it cannot be recorded on paper. Remember, IB moderators only see the paper – they never see the running program.

Some Suggestions

Before you give in and take the easy route of a data-base problem, consider some other possibilities:

Realistic Problems - Although not strictly required, a "real" or "realistic" problem generally provides more opportunities for the students 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 quiz/tutorial for a school subject (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)

Graphics – Graphics make things more interesting. There are plenty of images available on the web. But avoid animations (cannot show these on paper), and avoid highly technical work, like image filters – unless you are a very strong math student. How about a math graphing program? Graphics can be a part of any solution, like including some nice pictures, as long as they are not animations.

Some Warnings

Parents' Business - Many students try to do something for their parents, especially their parents' business. Parents are willing helpers and generally available. The student might even have a working understanding of their parents' business. Unfortunately, real businesses have computing needs that generally exceed a student's programming skills. And some businesses have legal or ethical restrictions that don't really allow them to use a program written by a student.

Interesting Problems - Many "interesting" problems, such as E-mail, playing chess, and video games can be unsuitable for the because:

- the problem is too difficult (for the same reason it is interesting)
- interesting problems can be 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. They must ensure the topic has reasonable scope for complete coverage of the required **mastery factors**, 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 problem and solution.

Unlimited (?) - 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. Rather than writing a word-processor, the student could create a program that analyses essays by counting words and producing an **index** of **overused** words. Even if this is only usable on text-files (not .doc files), it fills a real need without being too large or too complex.

Still**, the solution** should not be overly simplistic. Keep the problem relatively simple, but produce a really good solution.

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, but they 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 similar users, but this should be a small group. "All teachers" is a bad choice, but "several teachers in the math department" is okay. "All students" is a bad choice, but "some of the IB Diploma candidates at my school" is better.

- Think about the intended user when **choosing the problem**. Making a "personal calendar" is different for a businessman than for an elementary school student.

- 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 user discussions.

- 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").

Mastery - is the problem hard enough?

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, other problems are possible, and 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 lots of results, such as a simulation producing output every minute for 24 hours, 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. The validation routine might permit adding new keywords, like a spell-check that permits changing 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, providing "contextual" help.

- **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 case, the resulting data-file must be manipulated and used in a meaningful way .

Stage A Step by Step 1 to 2 weeks (5-10 hours)

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. Follow these steps – and **take notes!**

~ 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.

~ Find a User ~

Find a **user** who is interested in the idea. Collect **information** and **sample data.** Help the user to describe **tasks** and **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.

~ Initial Design ~

Make a data-flow diagram showing data-files and user-interfaces, and outlining typical interactions. Use PowerPoint or a word-processor.

~ 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 basis for discussion than an actual commitment to a specific appearance or functionality.

~ Goals Meeting ~

Show prototype to user. Check that the prototype is consistent with the scenarios. Sit together and write a list of **goals**, **features**, and **limitations**.

~ Criteria for Success ~

Write **Criteria for Success** based on mock-up and goals. This document serves a central role throughout the project. Be sure to include:

- Features - especially automation
- Usability requirements
- Reliability requirements
- Limitations

Get user approval and supervisor approval for these criteria.

~ Supervisor Approval ~

Obtain supervisor approval before proceeding to the design phase. The supervisor should 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.

Examples - here are some links to sample dossiers, ideas, etc:

Starting Simple: http://ibcomp.fis.edu/projects/quickAndSimple.html

Topics for a School: http://ibcomp.fis.edu/projects/TopicsForMe.html

TicTacToe Prototype:

http://ibcomp.fis.edu/projects/newproj/TicTacToePrototype.ppt

Adding Machine Prototype:

http://ibcomp.fis.edu/projects/newproj/AddingMachinePrototype.ppt

Stage A Checklist: http://ibcomp.fis.edu/projects/stageAchecklist.pdf

Average Stage A: http://ibcomp.fis.edu/projects/onlinegrades/analysis.pdf

Better Stage A : http://ibcomp.fis.edu/projects/newproj/DocsAF.pdf

SL Sample Dossier : http://ibcomp.fis.edu/projects/quizTacToeDocs.pdf

HL Sample Doss.: http://ibcomp.fis.edu/Projects/StudentSample2007.pdf

Stage B – Detailed Design

For Stage B of the project, you must produce a plan for :

Data Structures , **Algorithms** , **Modules**

that you will be need in your program. Start by reading the **goals** and the **user-stories** from Stage A. Look at specific words and then:

(1) Nouns for **Data-structures (Variables)**

NOUNS represent **data-items** or **data-structures.** The little things are data-items. Collections or lists are **data-structures.** For example, a **name** is a simple variable, but a **list of names** is a data structure - probably an **array**. If a list is **permanent**, it needs to be stored in a **file.**

(2) Verbs for **Algorithms (Methods)**

VERBS represent things **to do** - **algorithms.** When you identify an algorithm, try to link it to a data-item or data-structure. For example, rather than "search", it should be "find a **name** in a **file**" or "find a **name** in the **names array**". Try to group the algorithms together with the data-structure(s) affected in the same **class**. When possible, include **parameters**, **pre-conditions, post-conditions,** and **results** (return values).

(3) Modules (Classes)

Once the **data-structures** and **algorithms** are identified, **GROUP** these together into **cohesive modules (classes)**. Cohesive means the ideas belong together. In a program for a video-rental store, the **customer database** should be separate from the **video database**, and these should form separate **classes**. The **task** of **renting** a video will use both of these classes, but it should be in a third class - e.g. the **cash-register class.**

What Order to Do It

You can develop these ideas in any order. You might try to list all the data-structures first, then add the methods to the data-structures, and finally group data-structures into cohesive classes. But you probably won't think of everything the first time around, so you will go back and add more.

If you can think ahead, and **object-oriented programming** (OOP) makes sense to you, you might start with the classes rather than starting at the data-structures end. Classes correspond roughly to **areas** of the problem/solution. In the video-rental store, the customers are clearly a separate area from the videos, so it is easy to think of a **customers** class and a **videos** class.

In any case, you probably need several **iterations** in the development process – jumping around from data-structures to algorithms to classes. You can start by making simple lists without worrying too much about organizing the lists. Then take your lists and start grouping things together, putting algorithms together with data-structures and grouping them roughly into classes. Your **prototype interfaces** might help you with "discovering classes". Once everything is listed and grouped, start adding details - parameters for algorithms, sample data, pseudo-code for algorithms, and diagrams for data-structures and class relationships.

Most people will think about many things at once - data-structures, then algorithms, then back to data-structures, then invent a class, etc. Most students will find it productive to set a **clear goal**, like "list all data-structures" and **write something down.** Then work on another goal. Write things down but DON'T start programming. If you try to "do it all in your head" you are likely to get lost and confused, and waste lots of time "starting over". Be sure to produce lots and lots of rough notes, and **keep them** to include in the final documentation.

Attendance Example

Extracting Words

Start out by **highlighting** nouns and verbs (two different colors) in the user stories and goals, as shown below. This is a preliminary step which need not appear in the finished dossier, so feel free to do it on paper. (You might want to include these rough notes in an appendix.)

Bold = **noun** → variable (or class)

[brackets] = list → arrays and/or files (or classes)

italic = verb → task or method

CAPITALS = values → constants (final in Java)

Taking Attendance ***User Story***
At the beginning of each **class**, the [teachers] *take attendance*. They *mark down* the [names] of any [absent students]. If a **student** *arrives late*, they *change the attendance* to LATE instead of ABSENT. This is messy when done on paper – requires using pencil and an eraser.
Input – name of **absent student**, **period** of class meeting, **name of class**
Processing – At the end of the day, [names] of [absent students] must be *sent to the office*. This is currently done on a **standard form.**
Output – [list of absences] *sent* to **office**

Excuses ***User Story***
When [students] *return* from absence, they must *bring* a **signed note** to the **office**. The attendance clerk then *writes* EXCUSED next to the **student's name** in their [book of absences]. If a **student** forgets the **note**, the clerk *calls* the [parents] at home. If the parent says the student was not sick, the clerk *writes* UNEXCUSED.
Input – note or **message** from parents
Processing – *decide* whether the **absence** is excused or not
Storage - *write* "excused" or "unexcused" in the paper register

Variables, Data-Structures and Methods

Remember – this is not a complete dossier. Your dossier will be **much longer.** The example only shows two of the stories from the analysis, but you should do all your stories (and goals).

Now we can **extract** ideas for variables, data-structures and methods.

- Nouns = Variables
  student name, teacher name, class name, excuse, date, excuse, absence (String containing fields),
- Lists = arrays and files
  names [file], teachers [file], absent students [array], list of absences [file], students [file], book of absences [file], parents [file]
- **Verbs = tasks and methods**
  take attendance, mark down, arrives late, change attendance, decide excused or unexcused, write excuse

Classes from Prototype

The **user-stories** are usually about the **existing system.** The solution will include **new ideas** that make things easier and more efficient. The **prototype** was a first attempt at designing the solution. So we can extract more useful ideas from the prototype, especially methods and classes. An interface screen is usually a class, as well as complex data items.

- **More Methods**
  select class, mark absent student, delete absence (late), save absences, view student, enter excuse, see today's data
- **GUI Interface Classes**
  Take Attendance Interface, Clerk's Interface, Daily Summary, Excuse Entry Screen
- **Data Record Classes**
  Absence Record (Name / Class / Teacher / Date / Excuse)

Program Structure

Variables, data-structures and methods are only pieces of a program. A working program also needs a sensible **structure.** The general term for structural elements is **modules**. Classes, arrays and files are all modules in this sense. Inside a single class, **methods** break up the class into pieces which we can also think of as modules.

OOP (Object Oriented Programming) depends on **objects** as the major structural concept. A file is represented by an object – a BufferedReader or RandomAccessFile. An array is also an object, created with the **new** command. Complex data collections, like the data for a single absence, can be stored in an object containing a variable for each field.

Although all the modules in your program might be classes, you still need to distinguish between the **functions** of these various objects. There is no "right" way to structure a Java program, but here is one possibility.

- **(Teacher Interface Class)**
  # GUI Components #
  Classes List, Students List, Absent List, Save Button
  [Data-Structures]
  [absencesArray] , [absencesFile]
  ~ *Methods* ~
  chooseClass , *showStudents , chooseStudent , addAbsence , deleteAbsence , saveAbsences*

- **(Clerk's Interface Class)**
  # GUI Components #
  StudentName , AllDataButton , EnterExcusesButton
  [Data-Structures]
  [allStudentNames Array]
  ~ *Methods* ~
  chooseName , showAllData , showExcusesInterface

(This would be a much longer list for a real dossier.)

Presenting Your Design

Stage B – Detailed Design requires a design in 3 sections:

B1 – Data-Structures

B2 – Algorithms

B3 – Modular Organization

Some students want to work in a **top-down** fashion, designing classes first and then breaking them down as shown above (left). Others wish to work in the order specified, starting with data-structures, then adding needed algorithms, and organizing it all into modules as the last step.

Most people think about many things at once - data-structures, then algorithms, then back to data-structures, then invent a class, etc. Most students will find it productive to set a **clear goal**, like "list all data-structures" and **write something down**, then work on another goal. If you try to "do it all in your head" you are likely to get lost and confused, and waste lots of time "starting over". Be sure to produce lots and lots of rough notes, and **keep them** to include in the appendix.

However you do it, be sure to read the **Assessment Criteria** carefully and include all the **required** pieces. The following are **common mistakes** (possibly forbidden) from student dossiers that result in lost marks:

- **B1** – missing diagrams, missing sample data
- **B2** – *** code copied from the finished program instead of writing pseudo-code (forbidden!) ***, missing pre- and post-conditions
- **B3** – missing connections to data-structures

Sample Design for Attendance Application

The following examples are abbreviated (to save space). A real dossier must present ALL data-structures and ALL non-trivial algorithms, and show modules with connections to ALL the data-structures and algorithms.

B1 – Data-Structures

** Data-structures refer to arrays, files, records and Abstract-Data-Types.
** This does not include simple primitive variables – unless a String
** contains formatted data with several fields, like “Mon|p1|Math”.
** So individual ints, doubles, and Strings need not be described here.

** Remember that **sample data** IS required. Using realistic sample data
** makes the descriptions easier to write. Random data is less helpful.

Student Names

There are several functions that require a list of student names. Since names will be recorded in absence records, misspelled names would cause problems. So it's best to do all name inputs by selecting from a list.

All Students File

The program will start with a file containing the names of all the students in the school. The names will be written “Last, First”, one name per line, in alphabetical order. Each name will be followed by a list of **ClassID** fields, for each class the student takes. Hopefully these names and class Ids can be exported from the school's scheduling database, and transformed into a simple sequential text-file.

[AllStudents File]

```
Adams, Alice / Math 10a / English 10a / Science 10a / ...
Baker, Bobby / Math 10a / ESL 2b / Science 10c / ...
Bozo, Clown / Math 7x / English 7x / Science 7x / ...
Eagle, Eddie / Math 10a / English 10a / Science 10a / ...
......
......
```

This file will be accessed by various parts of the system, including :

- Teacher Attendance – populate a class list, e.g. Math 10a
- Clerk Interface – populate the Student Name box

Daily Absences File

For each class each day, the absences will be saved into a simple text-file, formatted as shown below. The files will automatically be save in a central folder on a LAN server, giving the office access to all the files.

[DailyAbsences-EinsteinA-01apr2008 File]

```
Adams, Alice / Math 10a / Einstein, A / 01 Apr 2008
Eagle, Eddie / Math 10a / Einstein, A / 01 Apr 2008
```

The clerk's module will open all the teachers' files and copy the absences into the AllAbsences file, where excuses will be added later.

All Absences File

AllAbsences is a RandomAccessFile containing an absence record for each student that misses a class. If a student is out sick for an entire day, there will be 6 absence records for that students on that day. Notice that on the day of an absence, the record will contain ??? as an excuse. This will get changed later – normally the next day.

[AllAbsences File]

```
.....
Eagle, Eddie  Math 10a    Einstein, A 01 Sep 2008 Sick Exc
Bozo, Clown   English 7x  Shakey, W   01 Sep 2008 Sick Exc
Eagle, Eddie  English 10a Cho, K      02 Sep 2008 Skip Unx
 .....
Adams, Alice  Math 10a    Einstein, A 01 Apr 2008 ???
Eagle, Eddie  Math 10a    Einstein, A 01 Apr 2008 ???
 .....
```

In a sequential text-file, it would be difficult to change the ??? - the program must load the entire file into an array, make changes, and then rewrite the entire file. A RandomAccessFile will be used so that changes are easier.

B2 – Algorithms

** Now that the data-structures have been designed, concentrate on the
** algorithms needed for those data-structures. Also design algorithms that
** perform actions indicated in the prototype - look at that again.

~ **loadClassList(classID)**

```
{
   pre-condition – class list files exist for each classID
   post-condition – name list box contains student names
   return – nothing
     --------------
   open AllStudents file
   loop through entire file
   {
      info = file.readLine()
      if (info.indexOf(classID) >= 0)
      {
         name =  parseStudentName( info )
         add name to Names list-box
      }
   }
}
```

~ **parseStudentName(info)**

```
{
   pre-condition – info contains a student record
                          name / class1 / class2 / ....
   post-condition – return the name field
   return – String name
     --------------
   return( info.substring( 0 , info.indexOf("/") )
}
```

** Notice that ~parseStudentName didn't come from a user story and it
** was not obvious in the prototype. It's a "low-level" service method,
** required to make the program easier to write. You will run into
** more methods like this as you are describing your algorithms.

~ **collectAbsences**

```
{  pre-condition – none
   post-condition – all absences have been copied into central Absences
                     file, and all the teacher absence files have been deleted
   return – nothing
     ---------------
   masterFile = open Absences central file
   String[]  files = createListOfFiles("server\absences")
   for (int f = 0; f < files.length; f++)
   {
       file = open(files[f])
       loop through file
       {
          absence = file.readLine()
          append absence to masterFile
       }
   }
}
```

~ **createListOfFiles(directory)**

```
{  pre-condition – none
   post-condition – all file names in directory returned in an array
   return -  String[]  fileNames
     -------------
    *** oooh, this is a tricky one – need to read the manual ***
}
```

** You cannot leave the tricky methods blank – you must actually write **
** something. It can be vague in some cases, but not missing. **

There are **lots** more methods needed - certainly all those named by extracting verbs from the user-stories. Very simple methods need not appear – for example, a method that asks the user which printer they wish to use. But students must describe all methods that contain **loops** or **complex logic** or **file access**. Most importantly, any methods that demonstrate **mastery factors** should be described, so teachers can see whether the student is going to demonstrate enough mastery factors.

Many students give up when writing the algorithms. Why?

- They have trouble thinking ahead that far, so they quit long before they have covered all the necessary algorithms.
- There is just too much to write – too many algorithms. This comes as a bit of a surprise, as students think of more and more algorithms each time they write an algorithm.

If the list of algorithms starts **growing explosively**, it's a hint that

- the problem might be too difficult
  OR
- the attempted **solution** is too complex and overly ambitious
  OR
- the student is programming inefficiently, adding new code for each new idea rather than writing re-usable methods. (see page 10)
  OR
- all of the above.

Unfortunately, by this stage it may be too late to choose a different problem. But it's not too late to design a more efficient and less ambitious program. As a famous scientist said:

“Make it as simple as possible – but not simpler.” Albert Einstein

** **Re-usable** methods make things simpler - **use** them lots and lots. **

B3 – Modular Organization

In OOP, modules are normally **classes**. Classes correspond to real-world **objects**. An OOP **Object** usually represents a combination of data and algorithms. These can be extracted from user-stories and goals. This could be done first, before data-structures and algorithms, guided by the Prototype and Goals. Or it can start with a very simple version and expand as more data-structures and algorithms are described.

A preliminary version can be written as an **outline** – this is easy to expand.

Teacher (teacher's interface)
uses #Students File, #Absence Record, #Daily Absence File
does ~loadClassList, ~markAbsent, ~markLate, ~save

Clerk (clerk's interface)
uses #Daily Absence Files, #Students File, #Absence Record, #AllAbsences File
does ~collectAbsences , ~loadStudentNames, ~enterExcuses, ~printDailySummary,

Students File
contains name/class1/class2/class3/... for all students

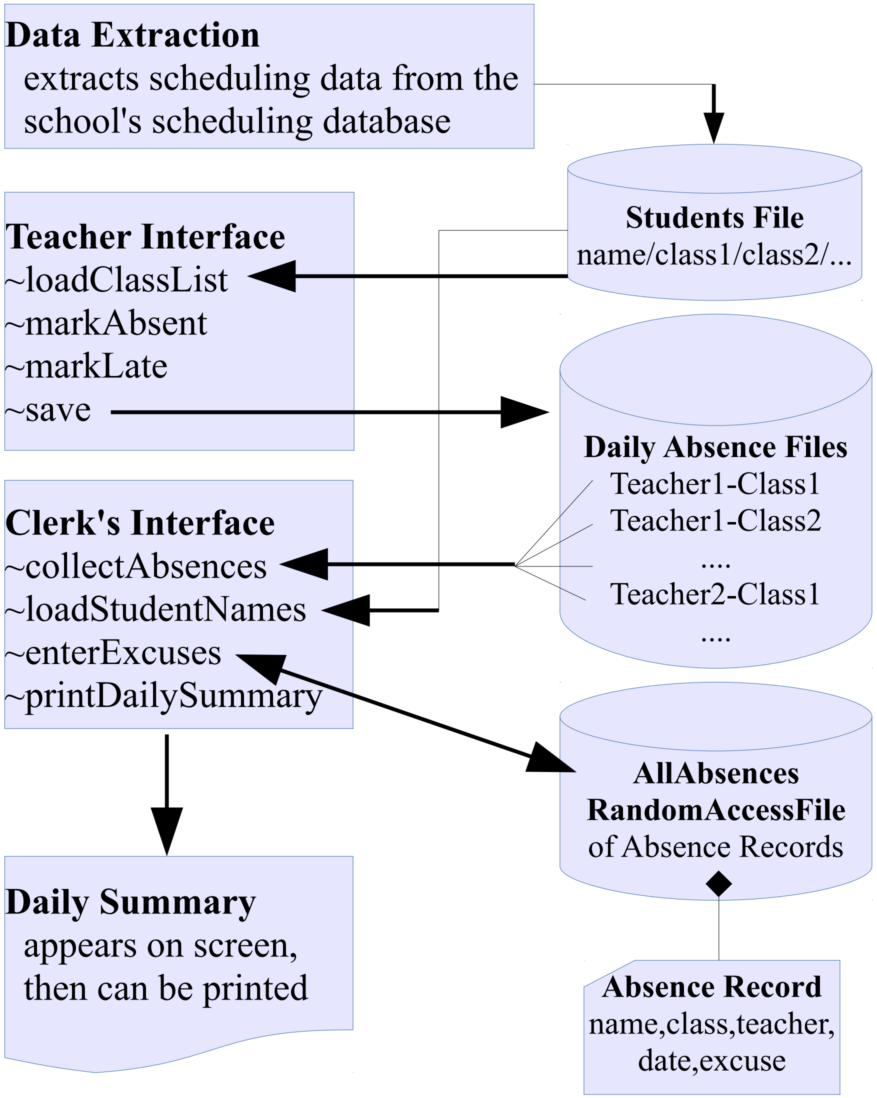
Absence Record
contains name,classID,teacher,date,excuse

Daily Absence File for One Teacher
contains name/classID/teacher/date

AllAbsences Master File
contains all AbsenceRecords for entire year and all students

For the finished dossier, a **diagram** is better than an outline as it gives a quick and clear **overview** of the entire design. This helps the student, user, and teacher (and moderator) understand the overall structure of the program.

B3 - Modular Organization Diagram



** Keep in mind this is only a simple, unfinished version – a real dossier
** would contain more information and be considerably more complex.

To receive full marks in B3, the student must have *"**described** all the modules, and has shown the connections to data-structures and methods."[Guide p 58]* So a description such as the following is required.

Teacher Interface

The Teacher Interface is used by teachers to take attendance during each class. It loads the names of students for the class from the Students file. The teacher marks absent students and then **saves** the data into a text-file for that class (one file for each class on each day).

Clerk's Interface

The attendance clerk works in the office. She will use the the Clerk's Interface for the following tasks:

- **Collect Absences** – opens all the files for individual classes and copies the data into the AllAbsences file.
- **Enter Excuses** – as students bring in excuse notes, the clerk decides whether an absence was excused or not and enters corresponding text into the record in the AllAbsences file.
- **Print Daily Summary** – prints a daily summary of all the absences for one day. The clerk will give a printed copy to the vice-principal.

This is really the central module for attendance management. The Teacher Interface is basically just an input module.

Data Extraction

This module must copy names and classes out of the school's scheduling database and store it in a format accessible by the attendance modules. It may be possible to do this from directly inside the scheduling system, in which case this module will not be part of the attendance program. If that's not possible, then something will need to be written here.

Stage B Step by Step 3 - 4 weeks (10-15 hr)

Item Extraction

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

Tasks Outline --> Program Outline

Write an **outline** of the **tasks** the user will be able to perform – (could become Menues). 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 mentioned

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

** **Supervisor Approval** ** Obtain approval before proceeding further.

Data-Structures

Design **Data-Structures** to accommodate data-storage needs of designed objects. Be sure to include **sample data** and **storage needs** (size), as well as **diagrams** to clarify non-trivial structures. Clearly explain files, arrays and ADTs, including the **reason** for using them. New ideas may arise 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

It's likely that students will go back and forth between Modules, Algorithms and Data-Structures, rather than proceeding in 3 separate steps. New ideas for **criteria for success** may occur - e.g. reorganizing files, use of other hardware, etc. These can be integrated into the original goals document, as long as important user-oriented goals are retained.

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.

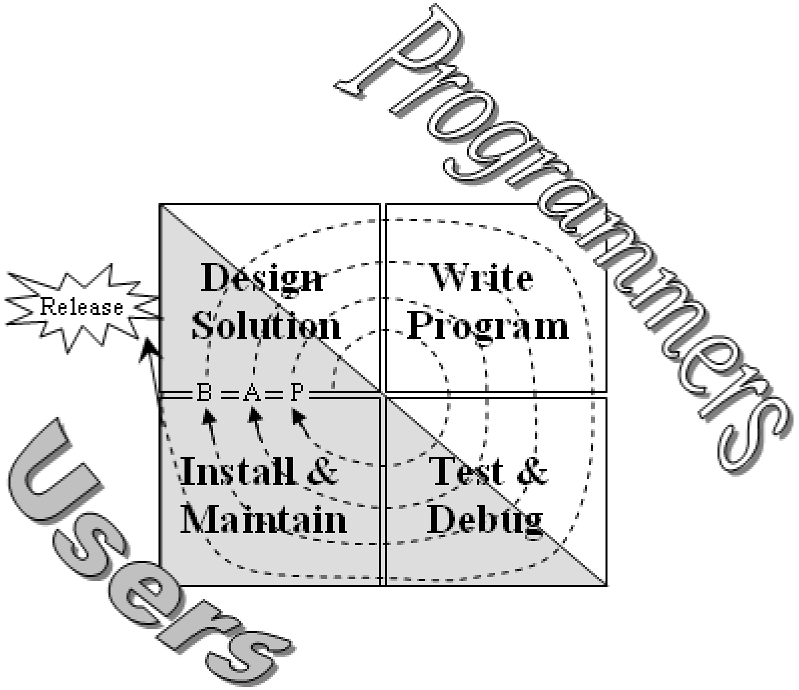
** **Supervisor Approval** *********
Obtain supervisor approval before proceeding to **programming**. The supervisor should ensure there is sufficient scope to achieve 10 (or more) **mastery aspects**. He/she should also ensure that classes, data-structures, and algorithms adequately support the goals stated in the **criteria for success**. The plan must be **achievable** within the prescribed time-frame.

Stage C – The Program

If students have learned Java programming for a year (or more) before starting the dossier, then writing the program should be relatively straightforward – though it probably takes more time than expected. A good design should make it easier and quicker.

Iterative Development

The following diagram demonstrates an iterative model for software development. It starts with a **prototype** and proceeds through several iterations (loops), including **alpha**, **beta**, and **release** versions.



This is based on Extreme Programming, which involves users directly in the development process. After each loop (version), the programmer should let the user test it.. This is a suitable concept for the dossier, where students should be working directly with the users.

Functional Prototype – Get Started

During Stage A – Analysis, you could have written a **functional prototype** rather than an **interface prototype**. A **functional prototype** is a very simple, limited version of the program that only implements a few of the features. This is tempting for students who want to start programming, but as one software engineer said: "Plan on throwing your (functional) prototype away – you will, so you might as well plan on it." This is distressing, so it might be better to wait until Stage C to make a prototype.

So what is a functional prototype? It should display a very simple interface, requiring only a few inputs, and then perform only very basic automated functions. There are no "nice" features, like drag-and-drop and pretty colors. There is probably little or no error-handling.

Stage C requires documentation of User-Friendly features and Error-Handling, so you will need to add these eventually - but you don't need to have them in the very first version. Below is a functional prototype for the Teacher Interface module.

```
import java.awt.*;
import java.io.*;
import java.util.Date;

public class Teacher extends EasyApp
{
   public static void main(String[] args)
   {   new Teacher();   }

   Label lbla = addLabel("Classes",50,50,100,20,this);
   List classes = addList("Math 9a|Math 10a|Math 10b|
             Comp Sci 1|Comp Sci 2",50,70,100,200,this);
   Label lblu = addLabel("Students",200,50,100,20,this);
   List students = addList("",200,70,100,200,this);
   Label lbls = addLabel("Absent",350,50,100,20,this);
   List absences = addList("",350,70,100,200,this);
   Button save = addButton("Save",500,100,50,50,this);
   String teacherName = "EinsteinA";
```

```
public Teacher()
{
   this.setTitle("Teacher Attendance Interface");
}

public void actions(Object source, String command)
{
   if (source == classes)
   {
      loadClassList(classes.getSelectedItem());
   }
   else if (source == students)
   {
       markAbsence(students.getSelectedItem());
   }
   else if (source == absences)
   {
      absences.remove(absences.getSelectedIndex());
   }
   else if (source == save)
   {
      String today =
        (new Date()).toString().substring(0,10)+" 2008";

      saveAbsences("d:\\attendance\\"
                + teacherName +"-"
                + classes.getSelectedItem()
                + "-" + today + ".txt");
   }
}

public void loadClassList(String classID)
{ // Stub version - this needs to load from disk files
   students.removeAll();
   if (classID.equals("Math 10a"))
   {
     students.add("Young, Al");
     students.add("Younger, Betty");
     students.add("Yung, Carl");
   }
   else if (classID.equals("Comp Sci 1"))
   {
      students.add("Jung, Fred");
      students.add("Older, Pops");
      students.add("Younger, Betty");
      students.add("Zonker, Zeke");
   }
   else
   {  output("Class ID is unknown");  }
}

public void markAbsence(String name)
{
     // check whether name is already absent
      boolean found = false;
      for (int c = 0; c < absences.getItemCount(); c++)
      {
         if (absences.getItem(c).equals(name))
         {  found = true; }
      }
     // if not yet absent, add name to absences list
      if ( found == false )
      {  absences.add(name);  }
}

public void saveAbsences(String fileName)
{   output(fileName);
    try
    {  PrintWriter file = new PrintWriter(
                 new FileWriter(fileName));
       for (int c = 0; c < absences.getItemCount(); c++)
       {
         file.println(absences.getItem(c));
       }
        file.close();
    }
    catch (IOException ex)
    {  output(ex.toString()); }
}
}
```

Improvements

Once the prototype is working, it requires significant rework and expansion. First, it needs lots of **comments** to improve **readability.** Keep in mind that an IB moderator will never see the running program, so they will be required to read the program to decide whether it satisfies **mastery factors.** HELP them read your program by using comments.

Mastery Factors

Students and teachers should already think about mastery factors during Stage B, to avoid producing a solution that is too simple. The design must provide enough opportunities to satisfy at least 10 mastery factors.

If an HL student designs a solution with no files, it cancels out 3 mastery factors. If the design also omits ADTs, it would be impossible to achieve 100% mastery.

Weak SL programmers might be tempted to write one long program in the **main** method, with no other methods. Then there are 3 mastery factors that cannot be satisfied. If there are also no arrays or files, there will also be no searching or sorting, and then it becomes impossible to achieve 100% mastery.

Are there enough mastery factors?

Since teacher judgment is involved, and not all teachers are the same, it's possible that a teacher awards 10 mastery factors, but a moderator only awards 8 or 9. The penalty for 2 missing mastery factors is 20%, which almost certainly lowers the final grade by 1 (or more), so this disagreement could be quite costly. The simplest way to avoid this danger is to plan on satisfying MORE than 10 mastery factors. In any case, all the mastery factors are techniques that make programs more flexible and more successful. So using more mastery factors probably has other benefits.

Are the mastery techniques "trivial" or not?

Keep in mind that mastery factors are only satisfied if they are used for a "non-trivial" purpose. That means the program actually benefits from the use of that technique. For example, an array containing one single item is unlikely to be beneficial. If the program writes a data file but never reads it again, then how is the program benefiting?

Common examples of **trivial** use are:

- a method returns a value, but the value is never used for anything
- an ADT has it's data changed directly without using internal methods
- loops are used inefficiently or incorrectly
- recursion is used where a loop would be more sensible
- a class extends another class but never uses the inherited members

Alpha Version - Functionality

The purpose of the prototype was just to get "something working". The next iteration is an **alpha** version – this is the time to get "everything working." Well, not really everything, but this should implement all the essential functionality, especially all the **mastery factor** algorithms. For example, all the file operations should function, sorting, searching, ADTs, etc. You might leave some **usability** improvements and some **error-handling** out at this stage.

This stage would require considerable reworking of the prototype.

- When it loads a class list of names, these must come from a file rather than having the names "hard-coded" in the program.
- It's not necessary to have ALL the class list files finished, but a representative sample should be present – at least 3 teachers and 3 classes for each teacher, and maybe 5 names in each.
- Absence data should be saved in the correct format, not just as a list of names

ALL the **modules** must be programmed – e.g. the clerk's interface, too. They must work to some extent, but not necessarily "perfectly". You'll probably want to add some error-handling – whatever is easy to include – but can leave the really detailed error-handling for the next version.

Spiraling Back

The development spiral looks good, like 4 straightforward steps. However, each loop might "iterate" several times until it's finished. That means the programmer might return again and again to the design stage, make changes, and then continue on the program.

There will probably be deficiencies in the original detailed design. It's common to discover (after finishing the design) some useful **low-level methods** that make your program simpler. For example:

- **validatedExcuseInput() returns String**
  This repeatedly asks for an excuse to be typed in. If the typed excuse is not in the official list {"Exc","Unx","Tdy", "???"}, then the method asks again. At the end, it returns a valid excuse.

    OR

- **chooseExcuse() returns String**
  Displays a dialog with a drop-down list containing valid excuses. The user chooses an excuse and this is returned.

Either one of these would do the job. Students seldom think in such detail that they design methods (before programming) for individual inputs.

It's acceptable to change the design **in parallel** with the programming changes. But it isn't actually necessary to have ALL the small, insignificant methods in the design document.

Note that it is **not permitted** to write the program and then copy Java code into the design document. So if a student wants to add something to the design – say a special sorting algorithm – they should **write a description with pseudocode** in the design, then use the ideas to write the Java code.

Slow and Careful

The prototype is probably a "quick-and-dirty" job, just to get started. Now the Alpha version requires slow work and careful attention to detail. As the program grows larger, debugging is increasingly difficult. A few tips:

- **Comments** help a lot. You will work on this program for several weeks. You will get confused. You will forget some of the things you did in the beginning. If you write comments as you go, it's much easier later when you look at the code and try to understand what you wrote.

- **Meaningful names** make the program easier to read – we call it "self-documenting". A variable named X might be used to count through an array. In a two dimensional array, you can use X and Y. But if you use ROW and COLUMN, it's much easier to keep track of what is going on (see style guide on next page)

- **Indentation** makes it much easier to see where **loops**, **if** blocks, and **methods** begin and end.

All three of these are expected for C1, so you will be rewarded with marks. But the big reward is finishing the program in less time.

Getting Unstuck

Student programmers get stuck sometimes. They change something in their program, it doesn't run correctly, and they **cannot find the problem**. DON'T stay up all night banging on the keyboard hoping to get lucky. Even worse – DON'T erase big sections of code and start over.

You **are** permitted to **get help from your teache**r. If you get stuck, spend 20 minutes or so trying to figure out what's wrong. After that, you should stop working and ask your teacher for help. Don't waste hours searching in vain. If you're at home, wait until the next day at school. Spend your time on something else – work on a different module, do homework for another class, clean up some documentation, etc. Don't get trapped in a stubborn cycle of unproductive, wasted time.

Naming Convention

The following web-site presents a recommended **Java Style Convention:**

http://geosoft.no/development/javastyle.html

Here is a summary of the most significant points:

- Classes have Noun names, and start with a Capital letter
  → TeacherInterface , AbsenceRecord
- Variables have noun names starting with a small letter
  → age , names[ ] , thisAbsence
- Methods have *VERB* names starting with a small letter
  → sortNames( ) , enterExcuses( )
- Constants are in ALL_CAPS, with underscores between words
  → final String UNEXCUSED = "Unx";
- Use CamelCaps for multi-word identifiers – capitalizeEachNewWord
- Local variables have short names
  → `for(int x = 0; x < 5; x = x+1)`
- "global" variables have long names (several words)
  → `String defaultDataPath = "d:\\attend\\";`
- Use get.. and set.. for data access methods in classes

There are lots more recommendations. The most important points are that the code should be:

- **Consistent –** use the same convention all the time. If you decide to name single variables with singular nouns, and arrays with plural nouns (int age , int[ ] ages), then do it that way all the time

- **Readable –** your code should be readable – for the teacher, for the moderator, and for YOURSELF. Use indentation, comments, understandable names, etc. You are expected to make your code as readable as possible. If you're not sure, ask your teacher to read it – they'll tell you whether it's clear or not. There are no rules, but the program must be **easy to read.**

C2 – Handling Errors

The dossier must document *"**many** error-handling facilities"[Guide p 60]*

For full marks : "*The student **fully** documents the error-handling of **each** input and output method within the program."[Guide p 60]*

This does not mean that every possible set of input data must be documented, but there should be numerous descriptions covering all the input and output **methods.** The student need not document the try..catch.. error handling in inputInt and other IBIO methods. But if a vaildatedInput method is written for a specific data value, this should be documented. The explanations can be brief. For example:

`Teacher.saveAbsences( )`
Uses standard try..catch.. commands to handle IOExceptions. When an error is detected, an error message pops up in an output window. This might happen if the user has removed an external hard-disk, or specified a non-existent directory or lost a network server connection. They can correct the situation and then press the [Save] button again.

`Clerk.inputExcuse( ) returns String`
inputExcuse uses a loop to input and validate an excuse String. If the user types a valid excuse in the list {"Exc","Unx","Tdy", "???"}, the value is returned. Otherwise they must try again. This guarantees that no misspelled excuses are saved in the AllAbsences file.

Some students like to copy Java code into this section to show how the error-handling works. That is acceptable but not necessary. In any case, written explanations are still required.

In some cases screen-captures might be helpful here, but generally the explanation is sufficient. This section may also use references (page numbers) to output provided in section D1.

C3 – Success of the Program

For full marks : *"The student includes evidence that the program functions **well**. The student successfully achieved **all** of the objectives from A2.....*

***Evidence** here refers to hard copy output in criterion D1....*

*The teacher should **run the program** with the student to confirm that the program functions, and that it **produces the hard copy output** submitted with the program dossier."[Guide p 61]*

This section does not need its own hard-copy output, but it might make reference to specific pages in D1.

The finished program should "function well." That means:

- it compiles without errors
- all features execute correctly (at least most of the time)
- the user can successfully use the program as intended
- there are few (or no) run-time errors

Many students fail to reach the "functions well" level, due to:

- starting too late or working too slowly
- attempting overly ambitious features
- sloppy testing

The Program MUST RUN – Keep It Simple

Often the program can be considerably more difficult than anticipated, making it difficult to meet the final deadline. Too many students submit dossiers with little or no hard-copy output as a result. Rather than submitting a non-functional program, it is better to submit a simpler program that runs – even if it does not meet all the objectives from A2. If it doesn't run, the student is penalized twice – in C4 and in D1. There may also be a penalty in the mastery factors because code that was supposed to demonstrate mastery actually does not work.

A **complete sample run** should show **all the features** of the program, executed as might happen in a single session or several consecutive sessions. For the attendance application, it could contain :

- two teachers taking attendance for several classes
- print out the resulting data files
- the clerk collecting the absences
- print out part of the resulting AllAbsences file
- print the daily summary report
- clerk enters excuses for the previous day
- print out part of the AllAbsences file showing the changes
- display one student's absences, showing excuses

If the goals include some sort of error-checking, or usability goals, then specific sample output must show that these goals were achieved. This may require thorough **testing** of the program, with correspondingly extensive sample output.

Documentation of the success of the program must include a list of all the goals (criteria for success) together with **sample output** demonstrate that the program actually achieved these goals.

The easiest way to connect the goals to the sample output is with a table that lists each goal together with a **reference number** (page number) to sample output in section D1. It is not necessary to duplicate all the output for sections C3 and D1 – but there must be a clear connection.

Remember, an IB moderator will never see the program running. So they can only judge the work by what is shown on paper, and this must be **complete** and **well organized.**

Stage C Step by Step 5-6 weeks(20-30 hr)

Incremental Development Cycles - Code, Test, Fix

Development should be cyclic. The Beta version is an expansion or elaboration of the alpha version. Each cycle includes programming, testing, and debugging. Comments in the source code should document the development.

Functional Prototype

This is just getting started. Make something that has a couple functioning features, just so you can get started and show the user something. This could be done in Stage A instead of the interface-prototype, but the interface-prototype is probably easier and more useful. Don't invest too much time in this – you might end up throwing it away.

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. Add more functionality. Debug some more. Accomplish all the difficult technical tasks - e.g. build the base of library functions necessary to make the rest of the program work This is a **technical** release, not a user release. But **DON'T** leave bugs in this version for later. The bugs are easier to find in this smaller version than hunting them down later in a big program. Make sure ALL the mastery aspects are accomplished in this version. The programmer should assess this version themselves.

**** Supervisor Check ****

Have the teacher check that all the **mastery factors** have been done adequately. If not, you will need to fix this in the next version.

Beta Version

Add any missing or incorrect functionality. Add or fix any missing mastery factors. Finish the user-interface and usability features. Add as much error-handling as you can think of. Make sure the program meets all the **criteria for success**.

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.

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 refer to **criteria for success.**

**** Supervisor Check ****

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

Stage D – Documentation

The main purpose of this section is to show the end-user, teacher and examiner what your program can do. Don't start too late. Make sure you finish your program in time to still do all the documentation.

D1 – Annotated Hard Copy

Since the IB moderator never sees the program running, all the information about the program must be printed on paper. This must include **comprehensive** documentation of **all** functionality of the program.

For full marks : "*The student includes a **complete** set of annotated sample output, testing all the objectives in criterion A2.*"

The sample output needs to be **organized** and **annotated** so that it makes sense to someone reading it. The following organization is suggested:

- A **complete sample run** showing **all the features** of the program, executed as might happen in a single session or several consecutive sessions. For the attendance application, it could contain :
  - two teachers taking attendance for several classes
  - print out the resulting data files
  - the clerk collecting the absences
  - print out part of the resulting AllAbsences file
  - print the daily summary report
  - clerk enters excuses for the previous day
  - print out part of the AllAbsences file showing the changes
  - display one student's absences, showing excuses
- Various **testing** of error-handling features
- Any particularly interesting **usability** features
- Organized testing of ALL **Criteria For Success** (from A2)

This sounds like lots of pages, and it is. The guide says "variable", but it's probably 30 pages or more. Many dossiers have too little sample output.

D2 – Evaluating Solutions

The Guide recommends answering the following questions:

> "• *Did it work?*
> • *Did it address the criteria for success?*
> • *Did it work for some data sets, but not others?*
> • *Does the program in its current form have any limitations?*
> • *What additional features could the program have?*
> • *Was the initial design appropriate?" [Guide p 62]*

For full marks : *"The student **discusses** the effectiveness and efficiency of the solution **and suggests** alternative processes and improvements.*
The student **suggests** alternative approaches to the solution and the design process."[Guide p 62]

Examples

Here are some on-line examples which show how to write stage D :

Complete Dossier http://ibcomp.fis.edu/Projects/StudentSample2007.pdf

Teacher Support Material:
http://occ.ibo.org/ibis/documents/dp/gr5/computer_science/d_5_comsc_tsm_0809_1_e.pdf

Stage D Step by Step 2 to 3 weeks (10-15 hr)

User Documentation

Before doing the final testing and evaluation, writing user documentation helps the programmer organize their thoughts about the program. It may also be useful to write some user docs **before** finishing the program.

FINAL Test Output

All the **testing and printing** 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 errors 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), sample data files with ample data, 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. These may also be referenced by sections C2 and C3.

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.

Final Comments: The 4 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 – not changing goals to make the programming easier.

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. 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 – avoid this whenever possible.

1 Guide : “Computer Science – First Examination 2006”,IBO, Geneva
2 XP : http://www.agilealliance.com/system/article/file/1006/file.pdf Extreme Programming the Oak Grove Way, Steve Mitchell, Oak Grove Systems
3 UML : http://www.visualcase.com/tutorials/use-case-diagram.htm Artisto Visual Case
4 McConnell : “After the Gold Rush”, Steve McConnell, 1999, Microsoft Press