DESIGN

3

B1 - Data Structures

B2 – Algorithms

B3 – Modular Organization

Stage B – Detailed Design

Introduction

In Stage B of the dossier, students are required to **design** their solution before **implementing** (writing) the program in Java. Most students find this challenging as they must think ahead and predict what will function.

Experienced software designers assure us that time spent on the design stage is not wasted time and that spending time thinking about the solution before actually writing it reduces the total amount of time required for the project.

Problems and decisions must be faced sooner or later – putting this off until the middle of the program is likely to result in false starts and insurmountable problems, causing the student to throw everything away and start again. So, a thorough effort in Stage B is worthwhile.

The IB assessment criteria seem to imply a traditional higher-level language – a 'top-down' approach – but this is not a requirement. An Object-Oriented Programming (OOP) approach is encouraged and may well start out with B3 – Modular Design, then move on to the details of data-structures and algorithms. The B1, B2 and B3 sections may be done in any order, and many students find it sensible to take turns on all three – design a data-structure, then design some corresponding algorithms, then package it up into a class, and then go back to another data-structure. No specific order is implied or expected for this work. At the end, all 3 parts must be finished before starting stage C.

Stage B consists of breaking the problem down into smaller and smaller pieces and organizing the pieces into cohesive modules. You can start by putting a few ideas down on paper.

Exercises

Think of two important <u>sections</u> that your program will need – these may become <u>classes</u> or <u>modules</u>.

....

....

Think of one data-structure (e.g. array, file, record,…) that belongs in one of these 2 sections (classes).

....

....

Think of two important algorithms (methods) that are needed to process the data in the data-structure.

....

....

Think of two other algorithms your program will need that will NOT process the data in this data-structure, and thus do not belong in the same class with this data-structure.

....

....

....

For example, consider the Home Library sample dossier. That solution needs the following:

- *Modules* for (1) inputting data and (2) storing data.
- The data will be stored in a file (*a data-structure*) and will be accessed by the storage module.
- The storage module needs *algorithms* to search for data in the file and delete old records.
- *Algorithms* for validating input and getting the current date belong in the input module, not in storage.

Decomposition - Discovering Data, Algorithms and Modules

For Stage B you must produce a plan for the Data Structures , Algorithms and Modules that your program will need.

Start by reading the **goals** and the **user-stories** (or use cases) from Stage A. Look at specific words and then identify:

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.**

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. Be clear and explicit. 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** and **results** (return values).

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.**

Examining User Stories

Data, algorithms and modules can be "*discovered*" by examining user stories. Here is a sample story from the QuizTacToe sample dossier. **Nouns** (bold) and ***verbs (italic)*** have been identified.

Various Types of Problems

I give small 10 minute **quizzes** *about twice a week. I'd like to use the same type of* **questions** *as I have on these quizzes. Those are short* **problems** *with either* **multiple choice answers***, or quite simple* **numerical answers***, or* **single word answers** *(like* ***fill-in-the-blank****).*

Input:	***copy*** existing **quiz problems** into the computer to be used in the **game.**
Processing:	**problems** are ***organized*** according to **topic.**
Output:	**problems** are ***saved*** in **data files** on the hard-disk.

Exercise

Copy a user-story or a few goals from your Analysis. Highlight the nouns and verbs in different colors.

Which Mastery Aspects?

The dossier requires students to demonstrate specific programming techniques – these are called 'Mastery aspects'. Keep in mind that mastery aspects must be 'well-documented and appropriate' and that 'non-trivial' means … 'the program benefits from the use of the aspect'. This is much easier to accomplish if you have a **clear reason** for each mastery factor you use. The reasons should be explored during Stage B - Design.

Choosing **appropriate** mastery aspects will depend on the topic, the user's needs and the Criteria for Success (goals), as well as the your programming skill. Below are some examples based on dossiers submitted by past students. They outline both good ideas and poor (inappropriate) ideas. These examples represent a student's thinking at the design stage. They are presented very briefly, when you write about your choices there will be a lot more detail about your thinking.

SL Library Book Checkouts

This is a simple version of a program to track books checked out from a library.

Good (Appropriate Use)	Poor (Inappropriate)
2-D Array – contains Title, Author, date, customer in columns. Add a row containing a record for each book that gets checked out, remove a record each time a book is returned.	**Objects as Data Records** – save today's date in a Date object.
Files – array will be saved in a file and loaded back into the array each time the program runs.	**1-D Array of Strings** – each String contains a Book Title and the Customer's Name, separated by '/'. Add a String each time a book is checked out.
Search – find a record when a book is returned.	**Search** – can search for a book to see whether it has been checked out.
Loops – used in search, loading and saving.	**Sentinel** – ask the user whether they want to quit, store the answer in a boolean flag.
If..then.. – used in search and various other places.	**Loops** – ask the user about quitting, over and over again, until they say "true" or "false".
Sorting – used to alphabetize list of overdue books.	**Selection** – check whether the user want to quit.
Nested Loops – used to save and load 2-D array as well as sorting.	**Return values** - the "do you want to quit" question is asked in a method named **quitting()**, and it returns an integer value that tells the number of times the user answered incorrectly.
User Defined Methods, Parameters, and Return Values - use these to break program into reusable code modules.	**Additional Libraries** – use a 3rd party Web-browser library to allow the user to surf the web.
	Methods – execute methods in the Web-browser library
	Parameters – in Web-browser library

Appropriate use of mastery aspects is closely related to the design of the solution. The good design above is very sensible, leading easily to a usable program. The poor version contains a number of questionable design decisions, leading to a solution that is difficult to use. The poor design also indicates the reasons for the mastery aspects, some of which are inappropriate and would probably not earn mastery marks.

Below is a brief summary of specific problems.

Mastery Item and Reason	What's Wrong with This
Objects as Data Records – save today's date in a Date object	Unless the date is used for something in the program, just storing it in a Date object won't show mastery. Anyway, it's preferable to create a user-defined class with more than a single field, like a class to store the title of a book and a customer name as a record.
1-D Array of Strings – each String contains a Book Title and the Customer's Name, separated by '/'. Add a String each time a book is checked out.	This might be OK and get the mastery mark. However, a 2-D array or an array of records would be considerably more sensible.
Search – can search for a book to see whether it has been checked out.	This might be useful and might get a mastery mark. However, a better design would search for user names, to check for overdue books. Of course, searching for a user will be difficult since the name is buried inside a String concatenated with the book title.
Sentinel – ask the user whether they want to quit and store the answer in a Boolean flag.	It appears the student is expecting 4 mastery marks based on one very small method. There are several problems with this, as described below.

Mastery Item and Reason	What's Wrong with This
Loops – ask the user about quitting, over and over again, until they say "true" or "false".	- a single loop (if it's the only loop in the program) is not enough to show mastery of loops. The same goes for a single use of a sentinel and a single if.. command.
Selection – check whether the user want to quit.	- the method apparently returns the wrong value. It should return a Boolean value, not a number.
Return values - the "do you want to quit" question is asked in a method named **quitting()**, and it returns an integer value that tells the number of times the user answered incorrectly.	All that said, it is very unusual for an SL student to write a program that only contains one single if command and one single loop. The search feature (above) almost certainly contains a loop and an if.. command. So the problem here might actually just be poor documentation of the mastery aspects.
Additional Libraries – use a 3rd party Web-browser library to allow the user to surf the web.	This might get a mastery mark if surfing the web is actually an important feature in this application. However, it sounds like it was simply added on so the student could use a library and get this mastery mark. The program probably doesn't 'benefit' from this.
Methods – execute methods in the Web-browser library.	This will not get a mastery mark because the mastery mark is given for **user-defined** methods (written by the student), not for executing existing methods.
Parameters – in Web-browser library	Again, the mastery mark is given for using parameters in a user-defined method, not in an existing method.

The worst part of the poor solution is that it does not use files. It's hard to imagine how this could function, unless we believe the program will simply keep running forever and the computer will never shut down. If the program does shut down, then all the data in the array will simply disappear. A week later, there will be no record of who checked out the books.

HL Text-File Index Builder

This is a program to index all the words in a text document, such as a novel. It will produce an index that lists all the words and their locations in the document, like you see in the back of a textbook.

Good (Appropriate Use)	Poor (Inappropriate)
Searching for Data in a File – searching for words in a text file.	**Add to RandomAccessFile** – text-document will be stored in a RandomAccessFile with a single word in each record (50 bytes per record).
Parsing a Text File – splitting text into single words, broken by spaces, commas, periods and any other punctuation (except apostrophes).	**Deleting from RandomAccessFile** – can delete a word from the RandomAccessFile.
Use of Additional Libraries – GUI interface including Buttons, TextAreas, Lists, Menus.	**Search in a file** – searching for a word in the file.
Use of 5 SL aspects - selection, complex selection, loops, files, methods, parameters, return values in multiple places in the program.	**Recursion** – binary search to find a word in the RandomAccessFile.

Good (Appropriate Use)	Poor (Inappropriate)
ADT (3 marks) – linked-list for each word containing all the positions of that word in the text-file.	**Two-dimensional Array** – in each row, there is a word and a list of numbers showing where the word is found in the file.
Encapsulation – linked-list will be encapsulated into a LinkedList class.	**Hierarchical-Composite Data Structure** – a class that contains a word and an array of all it's positions in the file.
Polymorphism – LinkedLlist class has multiple methods with the same name but different parameter lists, like search(word), search(word, starting position). Also multiple constructors.	**Encapsulation** – the hierarchical-composite class will be defined in a separate class, with private data items.
Inheritance – Entry will be a class containing a word and it's LinkedList, as well as methods for saving and loading the entry from file.	**Use of 5 SL aspects** - selection, complex selection, loops, files, methods, parameters, return values in multiple places in the program.
	Use of Additional Libraries – GUI interface including Buttons, TextAreas, Lists, Menus.
	Inheritance – program will extend EasyApp which provides easy constructors for AWT GUI.

Appropriate use of mastery aspects is closely related to the design of the solution. The good design above is very sensible, leading easily to a usable program. The poor design has chosen to avoid ADTs and text-files – maybe the student doesn't like programming these. There are several design decisions in the poor version that will lead to a solution that is difficult to use. The poor design outlines reasons for the mastery aspects, some of which are inappropriate and would not earn mastery marks.

Below is a brief summary of specific problems.

Mastery Item and Reason	What's Wrong with This
Add to RandomAccessFile – text-document will be stored in a RandomAccessFile with a single word in each record (50 bytes per record). **Deleting from RandomAccessFile** – can delete a word from the RandomAccessFile.	The concept here is inappropriate and will make the solution quite difficult to use. Presumably the original document will be a text-file. It is very unlikely to be available as a RandomAccessFile, so the solution will probably only work for a sample file created by the student programmer. Also, why would anybody want to delete a word from a document? That doesn't make sense. The student may be trying to avoid the problem of parsing text-files, but that is inappropriate. No mastery marks.
Search in a file – searching for a word in the file. **Recursion** – binary search to find a word in the RandomAccessFile.	Binary search only works in a sorted list. But the words in the file will surely be in their written order, not sorted alphabetically. Even if the student writes a correct binary search algorithm, the application will not benefit, as it cannot be used. Therefore no mastery aspects can be credited.
Two-dimensional Array – in each row, there is a word and a list of numbers showing where the word is found in the file.	This is OK and will probably get the mastery mark, as long as the array is large enough (enough columns) to accommodate a word with hundreds of occurrences. Otherwise it would cause an unnecessary limitation. A Linked List would be better, but it's not required.

Mastery Item and Reason	What's Wrong with This
Hierarchical-Composite Data Structure – a class (Entry) that contains a word and an array of all its positions in the file. **Encapsulation** – the hierarchical-composite class will be defined in a separate class, with private data items.	It sounds like this class is only added to get the mastery mark. It will probably not be used for anything, since the index is stored in the 2D array. If the code is just written and not used by the application, then it will not receive a mastery mark. But if the 2-D array is replaced by a 1-D array of Entry objects, then these mastery marks can be awarded.
Use of 5 SL aspects - selection, complex selection, loops, files, methods, parameters, return values in multiple places in the program.	This is fine. HL dossiers almost always get this mastery mark – as long as the methods are actually used for something.
Use of Additional Libraries – GUI interface including a couple of Buttons, TextAreas, Lists, Menus.	Fine. A GUI interface is fine for the additional libraries mark, as long as it is not trivial. It needs to be more than just a couple Buttons.
Inheritance – program will extend EasyApp which provides easy constructors for and AWT GUI.	No, just extending an existing class is not enough for this mastery mark. Normally a student would be extending a user-defined class.

As in the SL Library project, the basic concept here is inappropriate. This could make sense if the original text document can be converted to Random Access by using a feature in this program – a text-file parsing method. But if the assumption is that the user will somehow provide a RandomAccessFile, then the solution won't work.

More Than a Textbook Exercise

The dossier is more than a textbook exercise. It cannot be completed simply by pasting together code from practice exercises. The dossier must be **designed** to solve the problem in a sensible way. Students can certainly use code and ideas from practice exercises, but they must still design a sensible application and modify their code to fit the design. An inappropriate design might score high marks in the Stage B criteria, as long as the design is thorough and well documented – but it could still be penalized in the mastery aspects.

An inappropriate design usually results in mastery aspects that cannot be awarded and the penalty can be very large. Although mastery aspects are awarded on the basis of the finished program listing, students should start thinking about the mastery aspects during the design stage. It is very difficult to 'fix it' later when they discover the solution just does not work.

A few more sensible designs are outlined below. You should not copy these outright, but you may use them as inspiration for your own dossiers.

Mastery Intentions – HL Students

Do this during Stage B – Design. Write a brief outline of your intended program. Then for each mastery aspect, write a brief reason or explanation of how it will be addressed. Then ask your teacher for comments.

Outline of Intended Program:

....

....

....

....

....

Mastery aspect and explanation for using it:

Adding data in a RandomAccessFile	
Deleting from a RandomAccessFile	
Searching for specified data in a file	
Recursion	
Merging sorted data structures	
Polymorphism	
Inheritance	
Encapsulation	
Parsing a text file or data stream	
Hierarchical composite data structure	
Five standard level mastery aspects	
ADT 1. Incomplete 2. All Key Methods 3. Some Error Checking 4. Complete and Robust	
Use of additional libraries	
Insert into an ordered sequential file	