Summary

This project is about the research, design and creation of a graphical user interface to aid students in selecting and experimenting with their academic modules.

Research into different programming languages and how the current system for selecting academic modules was carried out. There are two main considerations with this project; one was the student's point of view for using the software and the other being the administrator's point of view for module data storage.

Acknowledgements

I would like to thank first of all Chris Gillespie my supervisor and Dr Kevin McEvoy my assessor for the extensive advice and help that they have given me in order to complete this project.

I would also like to thank the support staff for providing me with the resources I required.

Finally I wish to extend my thanks to everyone else involved in this project and especially my friends who help make this project a success.

Contents

List of Figures

Chapter 1

The Problem

This chapter gives an outline of the problem, the requirements needed in order to solve it and a schedule for the project.

1.1 The Aim

The aim of this project is to create an interactive software tool to aid students in choosing their academic modules within the School of Computing at Leeds University. The idea is to eliminate the need for the existing module selection forms within the school. A variety of programming languages and tools will be researched in order to discover which are the most appropriate to implement the software tool. Specific methodologies used when designing user interfaces will be researched and used. Finally, a minimal functioning, interactive software tool will be designed, implemented and tested in order to solve the problem. There is a glossary on page 45 explaining any of the technical words that are used in this report.

1.2 Requirements

The minimum requirements for this project are:

1. Research into how modules are selected and presented
2. Research into a variety of programming languages and tools
3. Research methodologies concerning user interface design and Human-Computer Interaction

4. Design a minimal functioning piece of software
5. Implement the software
6. Devise and write a software user manual
7. Test the product
8. Evaluate the product

1.2.1 Minimal Functioning Software

The final piece of software should allow a student to choose a combination of available modules and have a clear visual representation of the modules on the screen. The software will only be used at most twice a year by a student and therefore must be easy to learn and use. It would also be ideal if the software was portable across different operating systems to allow the software to be more flexible. The software will be linked to a data file containing the module information which an administrator may update each semester.

1.3 Further Enhancements

Here are listed many possible further enhancements. Most of them are way beyond the scope of this project but it is important to state what may be possible for this software in the future. If time allows for this project, a few of these enhancements may be implemented.

1. There may be the function of submission available so that the students do not only experiment with their selections but are also able to submit their choices. This may completely eliminate the need for any paper work for the student.
2. It may be possible to integrate the application on to the Leeds University web site (http://www.comp.leeds.ac.uk). This would allow students to experiment with module selection online.
3. There could be an option to view the details about each module to prevent students needing to refer to the module handbook. This information would give them an idea of what each module entailed.
4. The software could be further integrated with the existing module enrolment software of the school of computing. This would allow the student to not only experiment with module selection but also enrol for the following semester from the same interface.
5. Further study could involve the way in which modules are selected within other departments at Leeds University and even include other Universities.

6. After the selection of modules it may be possible to display a preliminary timetable for the selected modules. This may be helpful for students looking to choose electives and trying to find the available times. This would however involve much more input on the administrators part as timetables are constantly changing within the university.

1.4 Deliverables

The deliverables will include the minimum functioning software, the test for the product, test results data, the user manual and the report itself.

1.5 Schedule

The schedule is designed to be a set of guidelines and may change depending on the progress rate of the project.

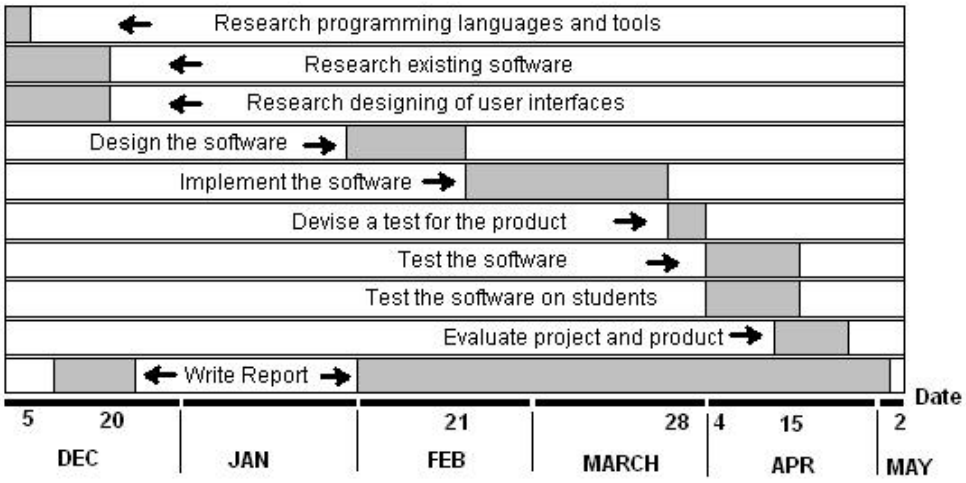

Figure 1.1: Schedule for the project

Dates of completion:

05/12 Research programming languages and tools
20/12 Research existing software
20/12 Research designing of user interfaces
21/02 Design the software
28/03 Implement the software
01/04 Devise a test for the product
15/04 Test the software

15/04 Test the software on students
25/04 Evaluate project and product
01/05 Complete Report

1.6 Conclusion

The minimum requirements of this project should result in a product sufficient enough to aid students with their module selection. If the project is ahead of schedule for any reason, a few of the 'further enhancements' may be implemented or discussed. The schedule is subject to change if any sections take less or more time than expected.

Chapter 2

Researching The Problem

This chapter will outline the research involved in order to discover the problem and the tools which can be used to solve it. Also, the chosen methodologies of user interface design will be highlighted.

2.1 Overview

With the current state of technology, and the fact that most things that were once recorded on paper are now automated, it was surprising to discover that the School of 'Computing' students still have to select their modules using a sheet of paper and a pencil. This project is about discovering how the academic module system in the school of computing works and to further more, to create some sort of software tool so that students may experiment with their selections.

Firstly, in this chapter, the module system will be researched and what is involved with the current module selection forms (See Appendix B). Further on in this chapter, the possible tools and programming languages will be researched to find which would be preferable to use.

2.2 Module Selection

2.2.1 The Pre-registration forms

Near the end of each academic year, with the exception of the final year, each student is given one module selection form. They are requested to use this form to select, from a choice, a certain number of optional modules to add to some compulsory modules. The first problem that any student faces is that they cannot

write on the form until they are sure of their choice. They could use a pencil and rub out any errors but this can still be untidy. The student is usually resorted to finding another piece of paper so that they can experiment with their choices. It would be preferable if this was not required by anyone.

Looking at this issue from the administrator's point of view, every semester they have to rewrite the module selection forms and reprint out a copy for each student. It is noticed that within the school of computing in particular, they emphasize the point that only one sheet per student is allowed to cut down on waste and to avoid reprinting extra copies. It seems clear that this type of software will over come this problem of wasted paper and wasted time on behalf of the administrator.

Module selection in the School of Computing is similar for each semester, degree programme and for each level(year). The difference between them is that the selection of available modules and the structure of selection varies also. The format is typically a set of compulsory modules followed by a further choice of modules which may be selected. These are called 'optional modules'. There is usually a limit of 120 credits which can be chosen (typically 10 credits per module). Each of the 12 modules have a set of prerequisites. These are modules that the student must have passed in order to undertake the option.

The choice of modules and prerequisites are usually portrayed on a module selection form (Appendix B). There is a separate form for each level and degree programme. The selections for both semesters are completed at the same time. One problem lies within the fact that the second semester prerequisites are not likely to have been achieved at the beginning of the year. For this reason, the software development tool will have to assume that any first semester modules chosen will be passed and classed as 'already achieved' prerequisites. This is to allow the student to pick second semester modules only if they have chosen the prerequisite module in the first semester.

Another issue is that any modules that a level one student may select, still have to be the correct combination in order to have the prerequisites for level 3. If they are not careful with their selection for second year, this may result in the student not being able to achieve the correct amount of credits in their final year(level 3).

Each module has a school(short) code (e.g AR21) and a university code (e.g COMP3300). The school codes have a history of changing as they come into line with similar modules. The university codes do not change as often. All codes should be visible on the interface as the short codes are designed to be easier for the students to remember and the university codes are important for further reference. If the school codes change and the titles, the tool should allow the data to be updated and will not require and program (code) changes. This is discussed further in the design chapter.

The main structure to the module selection varies slightly from year to year so it is important for the software to be flexible enough to adapt. For example, sometimes there are 4 compulsory modules and some times 5. This is discussed further in the chapter on design also.

The software will be aimed at students progressing to level 2 and 3. If more time allows, MSc (masters) students will be targeted. This should only involve the changing of the the data and not the software itself. Students just beginning the course do not have to select modules as they are all compulsory for level 1.

It should be safe to assume that anyone who has studied a computing course for a year will be familiar with the software components.

There is also the option to take electives which may include extra computing modules but are usually modules from other departments. Electives add a complication to the module selection process. The student may for example select 100 credits and have to find a further 20 credits of electives. There is an electives hand book by which the student may refer to to select from a very large choice of modules.

Current Method of Administration

Module administration throughout this project relates to simply finding a way to collaborate all of the module data for the forthcoming semester and place is in some sort of data file. It is clear that the administration that takes place for modules involves much more work than simply inputting the results but is simplified for the concerns of this project.

A previous method used in the School of Computing at Leeds University to create the module selection sheets, was to use a Microsoft Word file and simple type the sheets up manually. However, in the past few years there has been the use of a Microsoft Access report creator. This is a collection of several tables each holding the individual parts of the data. One table would hold the module codes and titles, another would hold the prerequisites and another whether it was a compulsory or an optional module. These lists are opened in different windows and the links between them take a while to discover. It is very important that a system is easy to remember how to use, more so in these circumstances were the file is only used twice a year. For this reason, this method is not ideal.

After these tables comes another window were the layout of each of the forms has to be manually drawn and updated. For example the titles are expressed and

the total number of credits have to be changed each time the number of modules change. It would be more efficient if a dynamic program was created which performed simple tasks like this automatically.

The compiled list of modules and data is also not available until a few weeks before the selection sheets are to be issued. This would therefore require any method of inputting the data to create the module selection sheets, to be a quick and easy process.

2.3 Graphical User Interfaces

There will be two sets of users. Firstly, the administrators who will input and update the module data. Secondly, the students who will use the interface to experiment with their module selection.

The most suitable methodology to use for designing the user interface is 'goal directed design' [4]. The theory behind this is that if the user's goals for using the software are met the user will be satisfied. If the user is satisfied the interface is successful.

The user's goals for this interface would be:

-Learn how to use the software quickly
-User's actions result in the expected output
-Successfully achieve what was expected and select their modules
-Have a resulting combination of modules which they can actually do (no invalid selections)

The interface will have to be clear and simple to use. It is important that the user does not become frustrated or need to spend too much time reading instructions or help files [12]. There should be the option of a user manual provided within the software and on paper in case anything becomes unclear. They should be able to locate the manual from a help file in the software menu.

The interface would also be more user-friendly if it behaves in a way that the user expects. An example being clicking on a clear check box will result in that box being filled with a cross (not something else being crossed or highlighted).
An important aspect of interface design is to prevent the user from making errors [4]. The interface will be designed so that the user makes minimal (if any) errors which will in turn keep them satisfied. A 'good' program will let the user walk away feeling that they have successfully completed what they wanted to do. That is one of the primary objectives

It would be preferable, if software is designed to be portable, for it to have the same layout on each operating system. Increasing familiarity of a program increases the user's confidence. They would not want to use the same software the following year on another platform and feel that it is a different program.

A major issue with software and user interface development a common mistake is made of using programming tools as design tools [4]. It is important that much care must be taken when mixing designing and prototyping. For the user's sake the design of the interface should be thought of from their perspective and not designed by what code can be implemented in the most simple or convenient form [7].

2.4 Tools and Programming Languages

2.4.1 Data Storage

The data required would include the module code, title, prerequisites and further information. These will ideally be stored in a single file for which the program can use. The data should be simple and quick to maintain and use by the administrator. This ruled out first of all the possibility of SQL or any other database management (DBMS) systems. There is the cost of learning extra software if DBMS are used. The administrator of the modules should be able to understand the database very easily, and therefore be able to update and maintain it with out having to learn any specific database application. This data file may also only have to be updated at most twice a year so must be simple for anyone to adapt to.

It is realised that there are many different advantages of using database management systems (DBMS). There would be greater data integrity. There would be stricter rules for the data to abide by resulting in less ambiguities and accurate data. However, for the purpose of this project, the simplicity of alternate methods outweigh the DBMS's advantages.

XML vs Text

There were two possible formats in which the module data could be stored which fit into the above criteria, considering the time available. The first is to simply store the data as a text file. The second is to store the module data as an XML file.

XML (eXtensible Markup Language) allows you to label aspects of data with tags (markup). Each of the tags can further be used to identify the data and text in XML documents [13]. XML's basic syntax is very similar to HTML but the purpose is different. HTML has a fixed set of tags by which limits the document's author. XML allows the creation of new tags to describe data more precisely [6].

One obvious advantage of this being that the tags can be called a name such that it is instantly clear to anyone what is contained within that tag. XML is a basically a simple, common format for representing structured information as text. [11]

An example of XML code is displayed in figure 2.1.

```
<?xml version="1.0"?>

<!-- Comment -->
<bankAccount>
 <person>
    <firstName>John</firstName>
    <lastName>Smith</lastName>
    <dateOfBirth>03/07/80</dateOfBirth>
    <accountNumber>00098461745</accountNumber>
    <referenceNumber>01446005</referanceNumber>
 </person>
</bankAccount>
```

Figure 2.1: Example of an XML file

It is clear from figure 2.1 what information is being stored. If anyone one wished to update a name or date of birth it would be a trivial task.

XML files and text files can be interpreted by all programming languages. As XML is self describing it is easier to handle [2]. Both of these text formats are are highly portable. The XML code is just as easily readable to humans as it is for computer program [11]. Text files are also easy to read but there is no obvious structure and the meaning of data may become ambiguous.

An example of how ambiguities may arise can be discovered when the above XML file example is displayed as a simple text file. This is illustrated in figure 2.2.

It is clear to see that a problem would be encountered using a text file like figure 2.2 when searching for the account number. There is an ambiguity as to which number is the account number. Is it '00098461745' or is it '01446005'? The problem is easily overcome with the use of tags, as shown in the XML file (figure 2.1.)

XML is classed as a mark up language which by definition is structure to a text file [11]. This will therefore make the database much easier to read and maintain

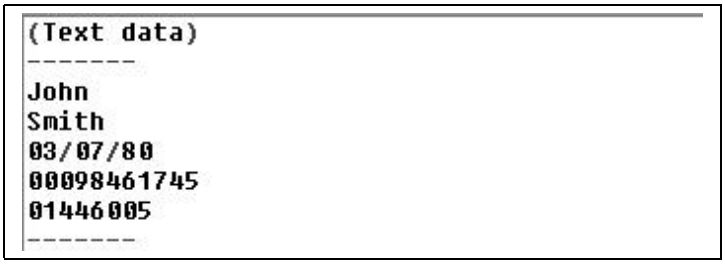

Figure 2.2: Data displayed simply as a text file

especially when it becomes large. Furthermore, not only is the XML file simple to read, update and understand but any program using the database file, would be able to call on the tags to locate specific data much more efficiently, as the tags act as 'landmarks' pointing to the data.

It is important for XML files to be syntactically correct for them to work. Even though XML files are very similar to HTML files, there are still further, more strict rules for which the XML file must abide by. Firstly, XML files are case sensitive (unlike HTML) meaning that the opening and closing tags within the file must match precisely in terms of upper and lower case letters. Another rule is that any opening XML tag must have a matching closing XML tag. To check if the XML file is syntactically correct, the file maybe opened in any web browser such as Netscape or Internet Explorer, any error will be displayed. Otherwise, a tree-like layout of the file is displayed.

'It seems likely that XML will become leading-edge technology for data representation' [6] and therefore when this project is complete, other universities and the rest of Leeds University may wish to use it for their module selection process. As the data will be stored in a universal format, many other programs may be written around this database. Unfortunately however, the School of Computing does not currently hold much of its data in XML form.

2.4.2 The Programming Language

Compiled vs Interpreted

A compiled language is a language that has to be translated into native machine code prior to any execution. An interpreted language is a language that is translated for use by the computer during the execution time of the program. Once code is 'interpreted', it can run on any platform and perform the same function.

C++ is an example of a compiled language. Once compiled, a C++ program will run much quicker than that of an equivalent Java program and takes up much

less memory. This becomes more noticeable in medium to large sized programs However, C++ can be platform dependant like most compiled languages and in some cases can make use of specialist libraries, which may be unavailable for different versions of the language [5].

It would be irrelevant for the compiling to take less time or even take up less memory because the interactive software tool in this case will be small in size.

Java

Java is one example of an interpreted language. Java is one known to be much more 'user friendly than C++' [10] and is less time consuming when it comes to making a program. One reason why Java has become popular so rapidly is that it can be used effectively for web page applets and applications embedded within web sites. Another major advantage of Java is that the programs are portable, as mentioned before, across many platforms.

Java has a rich collection of Java class libraries also known as Java API's (Applications Programming Interfaces). These classes are extremely useful in performing tasks, saving the programmer from having to code specifically many simple functions. A useful example of a Java API used within this project is JAXP (Java API for XML Processing). This can be used to extract and parse (process) information from the XML data file. JAXP in particular is discussed later in the chapter.

There are two main types of Java program. These are applications and applets. Applications can be stand alone programs that run simply as executables. Applets are generally designed to be embedded within a web page and used on the internet. Applications are more useful when the program becomes 'too large' to place on a web site. The latest version (at the beginning of this project) was Java 2 SDK Version 1.4.

Java Swing

Java Swing is another example of a Java class. It is a very simple and effective GUI(Graphical User Interface) toolkit. The GUI simply presents a 'pictorial interface' to a program. The toolkit consists of many common components such as buttons and text boxes which can be easily created and added to another component called a panel to create the GUI [13]. As Swing components are written in Java they provide a greater level of portability and flexibility. With the varying combinations of Swing components available, it is possible to reproduce almost any modern GUI available or any aspect of it.

The original Java GUI toolkit was called the Abstract Windowing Toolkit (AWT). This package consisted of very similar components to Swing. However, AWT did

not support a way of specifying a uniform 'look and feel' across different platforms. The Swing GUI components can be programmed to provide not only a uniform 'look and feel', but a different look and feel for different platforms or even the option to change the 'look and feel' while the program is running. Uniformity has the advantage of the user spending less time remembering which keystroke sequences or button clicks do what and spend more time in a productive manner [5].

There are many Java Swing functions that are dedicated to simply altering the layout and 'look and feel'. This has the benefits of being able to make the GUI look exactly how one wishes.

Perl and Python

Perl and Python are also examples of interpreted programming languages but do not have a simple GUI toolkit which can be learnt within the time allowed. Being partly familiar with Java before this project begun, it would preferable to consider mainly Java and it's APIs.

2.4.3 The Choice

An interpreted language would be more suitable for this project for the above reasons. Java has been chosen to be the programming language for which the interactive software tool will be written in. As time is limited and of great importance, it would be more useful to use Java as the selected programming language as it should be relatively easy to learn.

Java Swing will be the library (toolkit) used for creating the interface, combining the basic GUI components. The components which exist within the Swing library are sufficient and easy enough to learn for this project.

Although there has to be time put aside to learn the details of Java and Java Swing, in the long run, using Java will be less time consuming than using C++. A second reason is that it would be more beneficial for the program to be portable and flexible. The user interface could then be used on a variety of machines and operating systems throughout the university and at home.

Furthermore, future development is likely to embed the program within a web site, and unlike with C++, HTML is extremely compatible with Java applets. Java applets are not only ideal for use with web based programs, applets can be viewed separately also (by using an 'appletviewer' program). It seems that applets are the best option for the interactive software tool in the long run, and as a further enhancement, to embed it within a web page. However the program will firstly be written as an application for simplicity and possibly converted at a later date.

2.4.4 SAX, DOM and JAXP

SAX(Simple API for XML) and DOM(Document Object Model) are the most popular two different ways of accessing the contents of an XML file. They can be both used by many high level languages. The API acts as a link between the GUI program and the XML file. Their task is to read the data and convert (parse) it to a format that can be used in conjunction with another program. At least one needs to be used to extract the XML data.

SAX is by far the most complete and correct. There is very little that you can't do with it. SAX is an event based model which invokes methods when markup (e.g, a start tag, end tag) is encountered, making it very quick at accessing data. This applies especially to large data files. SAX is usually used when only reading is necessary from the file and the document does not need to be modified [8].

DOM on the other hand, is a tree based model that stores the data in a hierarchy of nodes. DOM has the advantage of being easily able to allow a program to write to an XML document. Another major advantage of both XML parsers is that "DOM and SAX interfaces for creating and manipulating (DOM mainly) XML documents are platform and language independent" [6]. DOM can be slow with large files as it has to scan the whole tree, however this will not be an issue with this project as the data file will be quite small.

JAXP (Java API for XML processing) is conveniently both of these parsers (SAX and DOM) bundled together into one. As long as the programming language used is Java (latest versions) then the use of JAXP seems to be the best parser to use. This option seems to fit nicely with this project.

2.5 Existing Software

It appears that there does not seem to be any software which has been developed concerning module selection within a university (in the UK). This may extend the possibility of using this software for other departments and universities. There are ways of automating the module data to be printed out in paper form, such as Microsoft Access' report creator, but there appears to be no program that displays them on screen as an interface.

2.6 Conclusion

With the cohesive nature of both XML and Java, integrating these two languages will produce an easily maintainable and reliable program, ready to use anywhere.

Also, it would be more ideal for the program to be written as an applet rather than an application as they are designed for small programs to run on the web (which is intended beyond this project). Finally, it is an obvious choice to go with the option of JAXP as the XML parser. This strengthens to choice of Java even further and leaves more options open as regards to the method of parsing (SAX or DOM style). Currently, the XML file will only be needed to read data from the file and not write to it. However it is an advantage to have the option available for future enhancements.

Chapter 3

Design

This chapter points out all of the aspects considered when designing this tool and results in a system specification.

3.1 Overview

The simplest option for this tool is to create a simple GUI (Graphical User Interface). The layout should be similar to the current module selection sheets (Appendix B).

After thought and consideration, there was a decision to have three separate files for this tool to be created. There will be an XML file which will contain the academic module data. Secondly, there will be the Java class that creates the graphical interface. To link these two files there will be a JAXP parser Java class. This class file will read the XML data and can be called by the GUI class and displayed.

A major consideration for the design of the interface was its ability to adapt to structure change within the XML file (see section on 'Dynamism'). This would make it more flexible and give the administrator more freedom in updating the file.

3.2 The XML file

The XML file will simply contain module information stored within tags, in a tree like structure. Having the data stored in tags will be a major advantage when it comes to calling the information into the GUI. The XML file should be easily

readable and understandable. This may involve instructions at the beginning and plenty of comments were required. It should be easily maintainable for anyone without prior knowledge of XML. The file will be designed so that an administrator would be able to easily update the information. For example if a module code changes or even if extra modules need to be added or removed. It is a priority for these tasks to be simple. Further information on how to update the XML file will be included in a user manual.

The information included in the XML file will be:

1. The COMP code of the modules
2. The short/school code of the modules (e.g CO33)
3. The title of each module
4. Each of the modules prerequisites
5. Information in detail about the module (e.g number of credits, semester)

The layout of the interface will be similar to the already existing module selection sheets and should also relate somehow to the layout of the XML file (to increase familiarity). This means that compulsory modules will be first in order of semester (one or two), then the optional modules will follow. It shall then be simple to relate the file on screen to the existing paper format increasing familiarity. The separate sections for each semester and type of module (compulsory or optional) should be easily located within the XML file, and labeled clearly.

The tasks that the administrator would have to do should be straightforward. If they wish to remove a module, they would be able to just simply delete or comment out the data for that particular module. To add a module they should be able to take a template, or copy one of the other modules as a template. With a template they will then only have to insert the correct data between the tags. Editing data within the tags should be self explanatory. An example of one module represented in XML format is displayed in figure 3.1.

3.3 The Design of the GUI

Java Swing includes many of the basic graphical user interface components. The components that are required for this particular interface will be text boxes, buttons, check boxes and a small menu. Each of the necessary components will be placed on a JPanel (Java Panel) which will act as the background to the interface.

First of all, the student will be prompted with a screen requesting which modules they have passed to date. This is important as the creation of the GUI will be dependant on which prerequisites the students have. Proceeding from this 'prerequisite request' screen, the program will reveal the main GUI.

```
<Compulsory_Module_Semester1>

»   <code>COMP3500</code>
»   <shortCode>PD31</shortCode>
»   <title>Professional Development III</title>
»   <prerequisite>PD13,PD14</prerequisite>
»   <info>
»   Professional Development III
»   Lecturer: Dr JR Davy
»   Credits: 10
»   Prerequisites: PD13,PD14
»
»   Assessment: Coursework 100%
»   »   »   »
»   Objectives: On completion of this module the
»   »   »   »   student will understand.........
»   </info>

</Compulsory_Module_Semester1>
```

Figure 3.1: An example of how a module is represented in XML format

The Layout

The layout of the main GUI will again be similar to the already existing module selection sheets. There will be uneditable text boxes containing the module titles, codes and prerequisites. There will be a button for each module which once clicked will display more detailed information about the module. This will eliminate the need for students to be constantly referring to the student handbook [3] to find out how many courseworks there may be or what exactly the module entails. Furthermore, there will be a check box for each module which will allow the student to select or deselect their modules. The compulsory modules will always be selected and displayed, similar to the existing module selection sheets. Near the bottom of the GUI there will be a box showing the total amount of credits selected so far.

The Rules

There are rules by which the student must adhere to when selecting modules. These are:

1. There is a maximum of 120 credits per year.
2. The student may only select a module for which they have the prerequisites.
3. If going into second year, they must select the correct modules in order to have a sufficient amount of prerequisites to gain entry to third year.

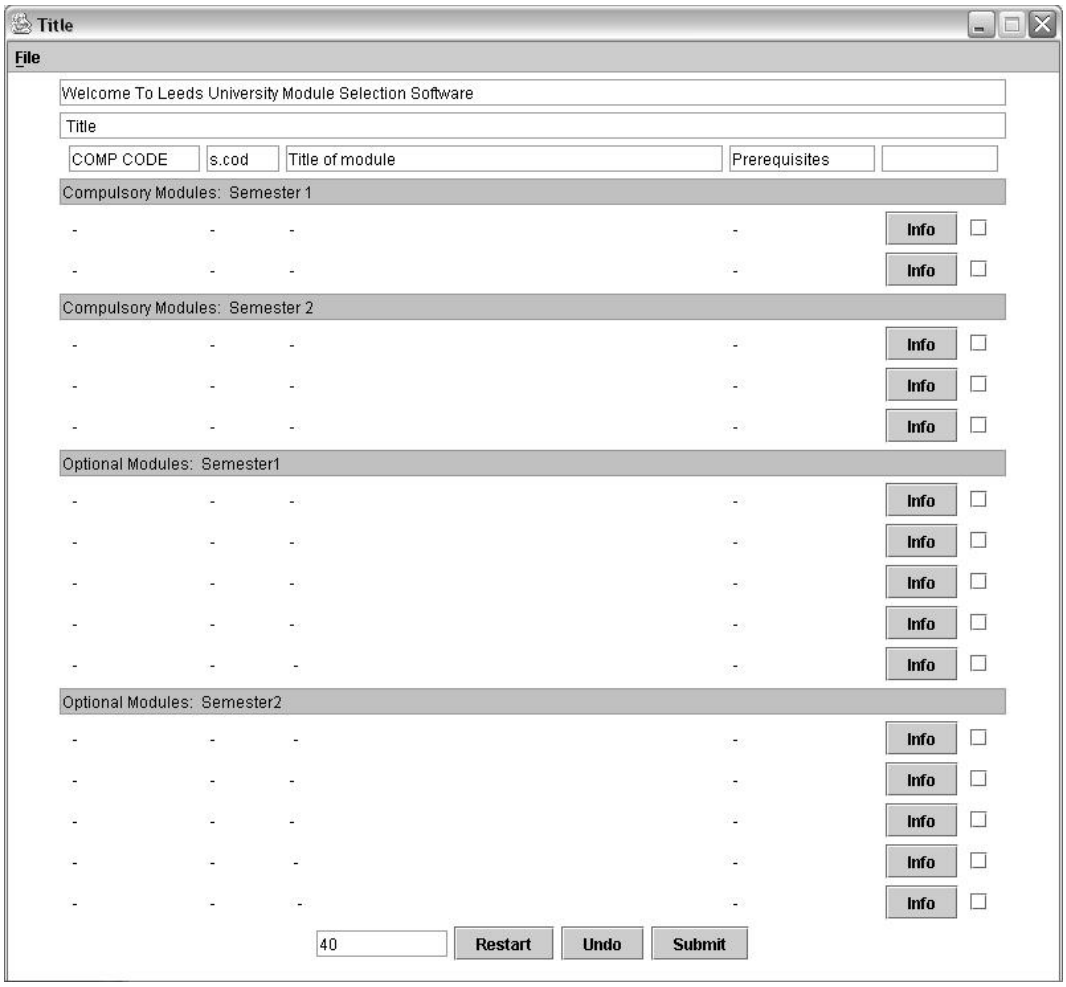

Figure 3.2: Early stages of the GUI

4. They must take the compulsory modules and these may not be deselected.

5. They must not exceed 70 credits per semester.

After having certain rules to abide by, the next consideration is how these may me implemented to avoid the student breaking any. For the rule: 'the students are only allowed a maximum of 120 credits', a simple warning would be necessary to tell them if they try to exceed this value. Another alternative solution would be to tell them when they have reached 120 credits any then they can the consider removing a selection before applying for another 10 credits.

There are several options for dealing with a student not having the required pre-

requisites. A method considered initially was to also give a warning when the student tries to select an invalid module. However, after further study it seems that a better solution would be to simply not allow them to choose the module in the first place. To do this the 'unselectable' modules should be highlighted to minimize trial and error on behalf of the student.

It will be assumed that the student will have the required prerequisites for the compulsory modules. These therefore will always be unhighlighted. This assumption is justified by the fact that the student must have passed these prerequisites in order to proceed to the nest academic year, otherwise they would not be using this module selection interface in the first place.

3.3.1 JAXP Parser class

The JAXP parser file will be written in Java also and will act as the link between the XML file and java GUI file. The JAXP parser will have the task of parsing the XML file so the attributes such as module code name and module title can be called into the GUI. The functions within this class will call the information from the XML file and then the main GUI file can access the information from here. This file is an important link between the data and the interface.

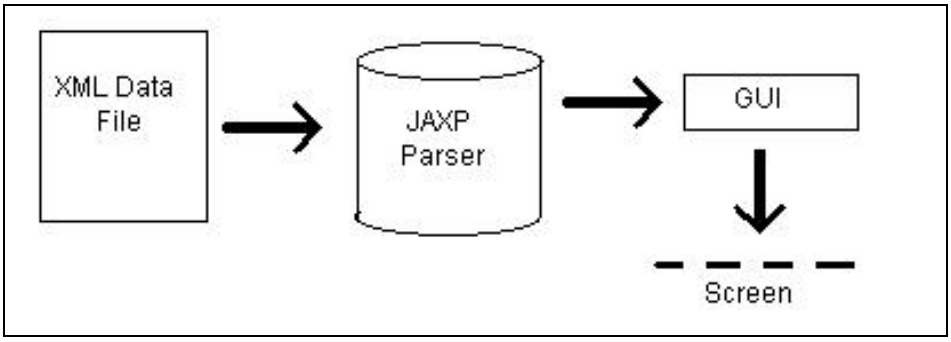

Figure 3.3: Diagram illustrating how each file is connected

3.3.2 Features

The GUI will have a menu which will allow the user to select from a variety of features. Menus are very common in GUIs and it is assumed the the user will be familiar with them. The menu will have a helpfile and an option to exit the program. Another possibility is to have a simple 'about' button which would display information about when the program was written and who by. The latter option however is not necessary, but adds to the professionalism of the product and is commonly found in many applications today.

Clicking on the helpfile within the menu will display the user manual. The user manual will be in two sections. One will include help on selecting modules and rules within the school of computing as regards to this. Secondly, there will be instructions on how to use the software (for both administrator and student).

There will be a button available on the GUI to allow a check on whether the student has selected enough prerequisites to allow them in to the next year. For example, a first year student has to select their second year modules so that they will achieve enough prerequisites to gain entry to third year. There will be a check on this possibility. If the student does not have the correct prerequisites they will have to change their selection.

After the student has completed their experimentation, it would be useful if they were able to submit their results in some way. This enhanced feature opens the possibility to many options. The student may submit electronically (by clicking on a 'submit' button). Secondly, they may save the file and be able then to email it, or they may have the option to print the results to a text file, listing the comp codes that they have selected, this could then be printed out onto paper and handed in to the university.

A simple scroll bar would be preferable as the list of modules becomes large, the user would need to scroll down the page to see more. An alternative to a scroll bar would be to use multiple windows each displaying different data such as one displaying second year modules and the other displaying third year modules.

For each module, the student may wish to know exactly what the module is about. A simple title will not suffice and therefore there should be some sort of option which displays information about the module. This may be a simple button placed next to the module which would retrieve the information and display it in a new window or dialogue box. The information should contain enough for the student to make a sound decision as to whether they wish to take that module or not. They will need to know what the objectives of the modules are, and what prior knowledge may be required to undertake each one.

Finally there is scope for an 'undo' button which would allow the student to go back one step. Undoing their actions would be useful if they select one too many modules and many not remember which they had selected last. Any warning message given for a student breaking any rules could be undone easily with this function.

Second and Third Year

As discussed throughout this report, there is the problem of selecting not only second year modules to the students preference, but to consider if from these choices whether they are eligible to take a minimum of 120 credits in third year. The current module selection sheets for first year students have both the second and third year choices attached together. They are required to consider their choices more carefully.

This somehow is a necessity to be incorporated in the design of this interface. There are several possible ways to achieve this goal. One way would be to display the second and third year modules in the same list. When a module is selected from second year, there is a presumption made that the student will pass it and it becomes an achieved prerequisite. If that module is a prerequisite for any third year module in the same list, then the third year module becomes selectable. Once all of the modules in second year have been selected, there will be a check that there are 120 credits selectable (available) from the third year. If so then the student may submit.

The student may be required to select also their third year modules as they may need the prerequisites from third year semester one to take third year semester 2 modules.

A simpler solution would be to have a button that checked automatically if they can take 120 credits in third year without the visual representation of the modules. This would be sufficient but not as effective or useful.

Aesthetics

The colour of the GUI is not to be overlooked. Considering carefully the use and application of this product, the colours used should give the impression of professionalism. The colours should be consistent throughout the interface itself and across different platforms. The text must be easy to read and not clash with the background. Studies suggest that "black text on a white background is always the most readable" Scharff, et al. (1996), mentioned in [1]. The font used will be standard and easy to read also. Borders around text boxes will be taken away to remove to keep with the current module selection sheets.

3.3.3 HCI Issues

Using common components to build this tool, it is expected that most people would be able to learn how to use it rather quickly. This is one of the goals of the software as the user may only need to use it once a year. The helpfile will be easily accessible for anyone who is unfamiliar with the components.

The tasks on the interface are to go no further than simply selecting and clicking with a mouse to complete what is required. It is proved that having the user to use the keyboard and the mouse to achieve different tasks takes longer, and is not efficient [7].

3.3.4 System Specification

The design of the system should be focused on in two main parts:

1. Usability of the Database File (administrator)
2. Usability of the Graphical User Interface (student)

The data storage should be simple and efficient. With the aid of the user manual and/or possibly a very simple tutorial, any administrator should be able to update modules, add modules and remove modules from the system.

The Graphical User Interface should provide satisfaction for the student after use allowing them to achieve exactly what they intend. This is to experiment with their modules and then submit their choices. Any possible help should be provided electronically and as a user manual on paper.

3.4 Conclusion

A simple Java Swing user interface will be created integrated with an XML file used for data storage. The administrator will be able to update the XML file, and the student will be able to quickly learn how to use the GUI and achieve experimentation with their module selection. There will be rules to be kept to by the student and many features to help them with their selection and submission.

Chapter 4

Implementation & Testing

The first part of this chapter outlines the main aspects of the tool's implementation techniques, and the problems that arose from them. Secondly, the chapter illustrates the tests devised and the evaluation of the tests. It is quite important for both implementation and testing to be considered together as test results lead to analysis of implementation (and design).

4.1 Aspects Of Implementation

4.1.1 XML file

It was a priority for the XML file to be very simple and easy to maintain. This required the name of the tags being in clear English with minimal abbreviations as possible. Originally, a 'compulsory module semester 1' was a tag called 'CMod1'. It was decided that this was very unclear and not easy for any first time administrator to remember. The tag for this section is now renamed exactly 'Compulsory module semester 1' for clarity. Similar changes were made to the other tags until they were all very precise.The final consideration for the XML file was that the modules stood out as separate modules with the use of spacing and standard indentation. A simple but practical solution.

There were a few other problems considered with this method of data representation. It was discovered that if the administrator does not keep the file in a consistent manner, it may cause disruption in how the information is displayed in the final GUI. If the administrator manages to miss out specific tags or have unintentional duplicates for example, the program will not run as required. As the program was designed with consideration towards dynamism, it has resulted

it in the GUI being highly sensitive to changes in the XML file. This issue should be pointed out to the administrator within the user manual.

One final after thought to the practical usage of this program, is that the XML data file has to be placed in the same directory as the executable program. This should also be specified to the user, and any updated XML files should be available for download from the university's website.

4.1.2 JAXP class file

The main reason for this 'linking' file was to read the XML data. There is standard code used in Java to read the file. At runtime, the XML file is check for syntax errors or ambiguities, if any are found the program halts and an error message is displayed.

The information from this file can then be used with a 'JAXP handler' in the main GUI class file. The 'handler' simply retrieves the data. The data includes the title of the course (header) and each relevant piece of data for the modules themselves such as the code, the title and the prerequisites.

The main problem with this program was deciding which method of parsing to use, as JAXP allows the use of many parsers (including DOM and SAX). In the final program, mainly SAX was used as being an event based parser, the program could simply react once certain tags were encountered. Another reason for choosing SAX was that there was no need to write data to the XML file while the program was running, if there was, DOM would have had to be used.

Another problem was the understanding of exactly what data to call from the XML file and how to do it. It seemed unclear, the following example explains the ambiguity that was encountered. An XML tag can be in the form:

$< tagvalue = "hello" > goodbye < /tag >$

It is confusing to understand that both 'hello' and 'goodbye', in this case, are called 'values'. With the Java function called getNodeValue() it would only return 'hello' as that is the 'real' value. To overcome this problem the tags would have to be put into an array and when in that state, the 'child node' is called using getChildNodes(). The child node contains the data 'goodbye'. From the child node, the function getNodeValue() is then called to extract 'goodbye'. It was difficult to see why there is no direct function to call 'goodbye'. The code to call the list of prerequisites is shown in figure 4.1.

```
public String[] getPrerequisites()
{
    String value[];

    Element root = document.getDocumentElement();
    NodeList Mods =
    // here the information is extract whenever a tag
    // called "prerequisite" is found in the XML file
      root.getElementsByTagName( "prerequisite" );

    value = new String[Mods.getLength()];

    for (int i =0; i < Mods.getLength();i++)
    {
    // all of the extracted information is placed
    // into an array of 'nodes'
    Node node = Mods.item( i );
    NodeList nodes = node.getChildNodes();

    value[i] = nodes.item(0).getNodeValue();
    }
return value;
}
```

Figure 4.1: JAXP method to call the list of prerequisites

Dynamism

The program is designed to be dynamic and as flexible as possible. This is an important factor when considering its use for other courses within the university, or even for other universities. It is dynamic in the sense that almost all of the information can be altered and the graphical user interface will adapt accordingly.

In order to adapt, there are functions within the JAXP class file which have the job of discovering the different lengths of the module sections such as compulsory module sections for semester one and two. This means that if the XML file contained five compulsory modules in semester one, there would be five boxes displayed for every module, each containing the correct information. This also allows simple information such as the title of the module (header) and the year it corresponds to (2nd or 3rd), to be changed to anything, even another title for use with other university departments.

4.1.3 Java GUI class file

When the GUI first runs, there is a first 'input' screen which requests the user (student) to input their already passed prerequisites in the short code form. For example 'co33'. They are entered one by one by the student and are placed by the program into an array (list) of 'prerequisites' which can later be called upon. When a student has finished inputting their prerequisites the main interface begins. Only the modules which the student has the correct prerequisites for become

```
if(completedPrerequisites[j].equalsIgnoreCase(preNames[i]))
{
// if a 'prerequisite' entered by the students matches
//a module in the data file
checks[i].setEnabled(true);// allow the check box to be ticked

//unhighlight the module so that it is clearly selectable
checks[i].setBackground(Color.white);
nameCourseText[i].setBackground(Color.white);
}
```

Figure 4.2: Unhighlighting of modules if the prerequisite is entered

selectable (and unhighlighted).

The user is allowed to input 'incorrect' data such as mis-typed words and the program will continue to simply ask for another prerequisite. The program is able to handle lower and uppercase text, treating them the same. This is implemented when checking if a certain prerequisite is in the array by using the java function 'equalsIgnoreCase()'. The 'ignoreCase' part is self explainitory. If the module requires two prerequisites then there are two checks made, to ensure that both are available before making that module selectable.

The code for this was written simply so that as the program runs, each item in the 'prerequisite array' is check and for each one, the relevant selectable modules become available. See figure 4.2

This code is to simply keep asking the student for a prerequisite and if the student does not enter anything, the program continues to open the the main GUI. There is a cancel button to exit the program.

The array of completed prerequisites must be updated once a module is selected, presuming that the student will pass it. the reason for this is as explained earlier, the possibility that they may not have the prerequisite for a second semester module until they pass it in the first semester. This has been implemented by, once the program is running, when a module is selected, it's short code is added to the list of prerequisites. If it is then deselected the code is removed from the array.

One problem encountered at this point was the realization that once the student had entered their prerequisites, they could not go back and change them or add more. The student would have to re-run the program and input them all again. Certain possibilities to overcome this problem were looked into, but none were implemented in the final program. A possible solution was a simple display of the

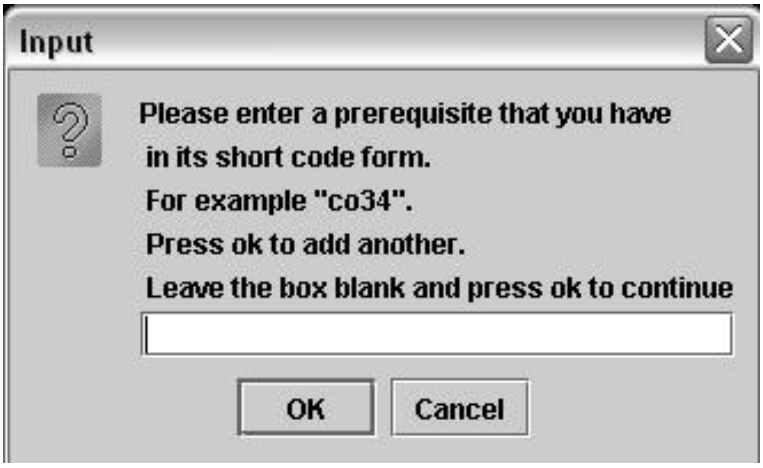

Figure 4.3: Input Prerequisite dialogue box

modules which had been input as a list, and a button to call up the prerequisite request screen again. This would then allow the student to input any they had forgotten first time around.

The user interface was implemented so that it looked similar to the already existing module selection sheets. With the use of the Java Swing API, simple text boxes, buttons and check boxes can be created with just a few lines of code. This can be extremely useful for developing this kind of GUI.

There are several different options as regard to customizing the layout with Java Swing. After experimenting with a few, it was decided that a simple layout method was to be used called 'flow layout'. This simply places one object after another in the order the code is written in.

Another sticking point arose when trying to keep the layout of the GUI consistent. There was a problem when the window was resized. The change in size caused the text boxes to move to different positions. This problem was overcame by implementing a fixed window size preventing the user from resizing and therefore the layout remains as planned. There was a possibility of having what is called a grid layout where by each component (e.g text box) could be positioned precisely at a location within a grid. This method would have taken up much more time in coding and after trying it out it did not look as neat.

If the student fails to input a prerequisite for any module, that module remains highlighted in grey and the check box remains unselectable. If they do have the prerequisite, the boxes are unhighlighted and the check boxes become selectable.

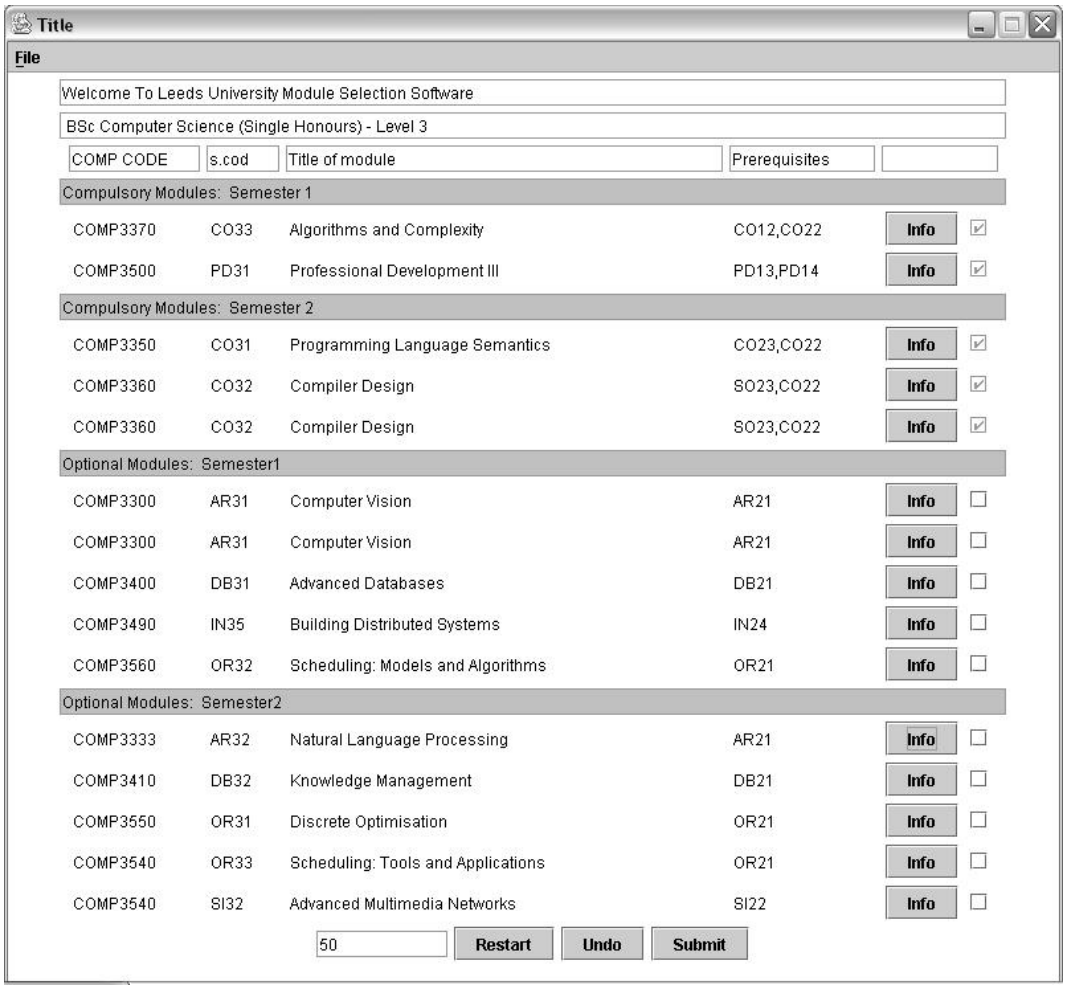

Figure 4.4: Main GUI layout

In the example 4.5 the module AR31 is unselectable because the student did not have AR21 as a prerequisite. However they can select DB31 as they must have entered the prerequisite DB21 (into the prerequisite request box).

The reset button was implemented so that it resets the total number of credits back to just the total number of compulsory module credits. It also clears each of the optional module check boxes, essentially putting the interface back to exactly how it was when it begun. If the amount of compulsory modules change, the reset button can still dynamically reset the total credits value accordingly.

As for each of the information buttons, when pressed, they use a function in the JAXP file to call the data contained within the $< info >$ tags for the corre-

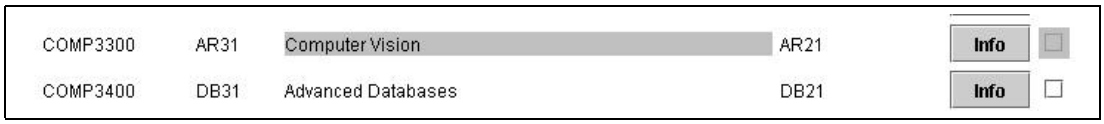

Figure 4.5: Showing an unselectable module

sponding module. The information displayed in a dialogue box would be in the same layout as in the XML file (including white spacing). This led to a potential problem of inconsistent layout of 'information' data. It would have to be a strict rule for the administrator to maintain the same layout within the XML file. If the layout in the given modules file is maintained then the information box will appear as shown in the example figure 4.6.
This information is taken from [3] and will take away the inconvenience of the student having to refer to it.

Submission

To begin the implementation of the submission function, the first step was to create a "submit" JButton (Java button). When the student is ready to submit their selections, they simply click on this button and it prints out, in short code, the selected modules in a list to a file called "submit.txt". With this file the student then has two options. They can submit the file via email and anyone receiving this file will know who it is from and will be able to enroll them. Secondly they may print ot the resulting text file and hand it in with their name on. This is a sufficient way to deal with submission as the file will be clear and easy to further administer for the person responsible for enrolling the student.

The implementation checks whether the student has selected the correct number of module credits (120) and that they are eligible for third year (if applicable) before allowing any 'write to file' function. If they do not have the correct amount of credits then they are issued with a warning and are allowed to select or deselect further modules. The code used for the submit button is displayed in figure 4.7. To deal with electives, the submit button allows the student to submit any selections less than 120 credits. If they do this, a warning is given and in the output file there is a section created reminding them to fill in any details about their electives. This is implemented by adding lines of text to the file if they submit with less than 120 credits. This is sufficient because otherwise, a huge database to cover each department and module would be required (to cover the electives handbook). This is beyond this project. It would be fine for the student to hand write their module information or add to the the email that they send of their choices.

The 'undo' button was never implemented in the final product. Research was

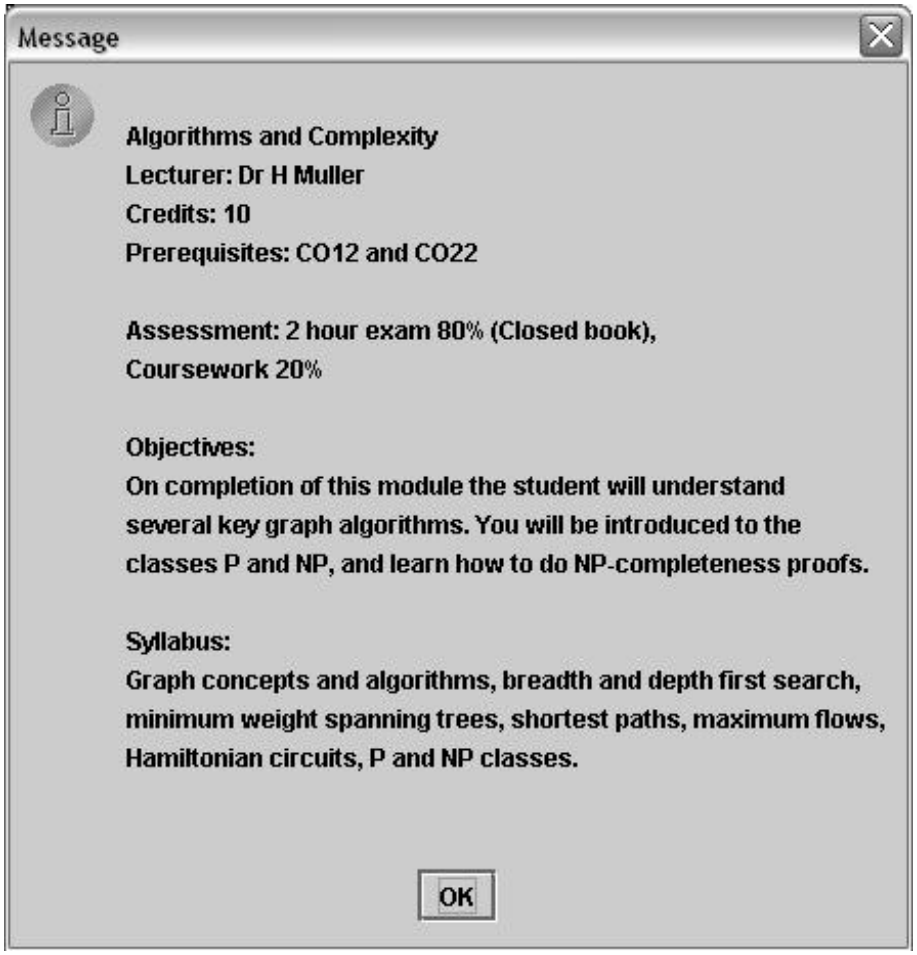

Figure 4.6: Information requested for the module 'Algorithms and Complexity'

carried out into how this may be done. Java has a relatively basic method of implementing undo (and redo) [13]. However, it was decided that this was an unnecessary function for the purpose of this application. Undoing an action would be to simply deselect the previous check box which had been selected, the user would probably rather deselect it themselves as this clarifies what actions have taken place.

Dialogue Boxes

Throughout the program, there has been extensive use of dialogue boxes. Dialogue boxes are simply 'pop up' windows with information and buttons at the bottom such as 'OK' to notify that you have read the information. Figure 4.8 shows what

```
if (e.getSource() == submit )   //if "Submit" is clicked
{
»   if (creditValue == 120)    //check if 120 credits have been selected
»   {
»   »   //create output file
»   »   PrintWriter fout = new PrintWriter( new FileWriter( "submit.txt" ) );
»
»   »   »   for (int i=0;i< totalLength;i++)
»   »   »   {
»   »   »   »   //if a check box is selected (state == true)
»   »   »   »   if (checks[i].getState())
»   »   »   »   {
»   »   »   »   »   //output the corresponding short code
»   »   »   »   »   fout.println(shortNames[i]);
»   »   »   »   }
»   »   »   }
»   »   // close output file
»   »   fout.close();
»   »   //notify the user
»   »   JOptionPane.showMessageDialog(null, "Your results have been printed");
»   }
}
```

Figure 4.7: Code for the submission button

a warning message dialogue box looks like, used in this program, when a student selects too many credits. In general it is not recommended to use many dialogue boxes unless they are necessary as they are known to break the flow of using the product [4]. However, there are three very different uses of the dialogue boxes used in this case.

Firstly, there is an input dialogue box at the beginning which is necessary for the

Figure 4.8: A 'Warning' Dialogue box

student to input their prerequisites. Next is the use of dialogue boxes as warnings, this is the most common use of dialogue boxes. It is important that flow is broken

when the user breaks one of the rules and can then be notified. Finally, one of the main uses of dialogue boxes within this program is instead of the use of a 'new window'. If the user requests information, it is displayed in a dialogue box instead of opening the information represented in a new window. This method has been chosen for simplicity within the code and it still results in an effective and practical method.

Third Year Issue

Some implementation was undertaken to try and display both the third year modules and the second year modules. The implementation was planned to show that when a second year module is selected, any third year module with that as a prerequisite becomes selectable in another window (or the same list). If there appears to be enough selectable credits for third year when the student is completed experimenting, they are allowed to submit their choices. This method of implementation caused many coding problems and another had to be devised.

The method used in the final product was a simple 'check' button which would run a check to see if they had enough selectable modules in third year, allowing the student to continue. The button would perform this query without any visual display of exactly what modules were available in third year.

The method implemented caused one problem. The student should also have to select their third year modules when they select their second year modules. The reason for this being is that they may not have a prerequisite to take a second semester third year module until they had selected the relevant prerequisite in semester one (of third year). This problem was considered during the implementation of the check button and the solution was simple. If the student becomes eligible for a third year module, the program assumes that they may then take this module, this then becomes added to the prerequisite list. The following example clarifies this:

Example: Three modules:

Second year module: SI22
Third year module (semester 1): SI32 (requires prerequisite SI22)
Third year module (semester 2): AB33 (requires prerequisite SI32)

If the student selects the second year module, they have the required prerequisite for SI32. The program assumes that they will take this module (because they can) and therefore get the required prerequisite for AB33. Simply from selecting SI22 in second year this opens up the possibility for two modules in third year. This method is acceptable because the result shows that they would be eligible to do enough credits in third year (even though they are the subjects they may not wish

to choose).

If more time was allowed for this project, there is no doubt that this problem would need to be rectified first and foremost. It would be more ideal to display both lists of options inside different tabbed windows.

4.1.4 User Manual

The user manual (Appendix C) was designed to be short, simple and clear. It has been written in two main parts. One part is for the administrator's use, designed to be a simple reference to how modules could be added, removed and updated with the assumption that the administrator had never used XML before.

The second part is a simple guide for the student on how to run the program and how to use it. It also includes the basic rules for selecting modules. This section of the user manual however, is not too detailed as the actual program is self explanatory and there are the relevant restrictions implemented so that the user may not violate any rules. It is not expected that the student will need to refer to the user manual, however it is provided incase.

4.1.5 Application to Applet

Currently, the program stands as an application. It will run on a command line basis like any other Java application. The earlier plan for this interface was to later convert it to a Java applet which would then allow it to be easily embedded within a web page. The resulting 'class' file which is called when running the program, was sufficiently small enough to be placed on a web page and downloaded in a few seconds at most. It was originally thought to be a trivial task of simply changing the 'main' function within the GUI class to an 'init'(initialise) function and that would work as this has work for simple java programs and appeared to be the difference between the two. However,after many different attempts it was found to be more than a trivial task. As a result of this the program remains as an application however with more time available it could be eventually converted into a Java applet for use with the internet. Learning from this, there is no doubt that if this project was done again, the program would have been written as an applet from the start.

4.2 Testing

This section shows the criteria devised to test the project on both the potential administrator(s) and on the students. The results are evaluated and discussed. Both, the administrator and the students that were tested were given a copy of the user manual to read through briefly or to refer to when necessary.

Further testing was done to try and uncover more bugs by trying different combinations of actions. This was planned to result in a debugging session, eventually ironing out or at least being able to highlight the problems with the code itself.

4.2.1 Test For the Administrator

One reason for testing the administrator was to decide whether there would be a need to develop a simple tutorial allowing them to learn the simple tasks.

The administrator will be required to do three main jobs:(See Appendix D)

1. Edit/update existing module information
2. Add a new module
3. Remove an old module

The tasks were set for a potential administrator within the school of computing. The administrator had no prior experience in using XML files or in writing Java programs. This makes the test fair and the results should show that almost anybody could undertake this test. The test was not extensive but the tasks covered in the test were sufficient enough to extract and analyse any problems.

Administrator's Test Results

The results of this administrator's test were very successful. Firstly on observation, the administrator managed to pick up the simple editing methods extremely quickly. There was minimal reference to the user manual other than a once read through at the beginning of the test. Each of the tasks were completed first time and in quite a short time considering the administrator had never seen the data file before.

The feedback from the test involved a discussion about where and how the data would be stored. The problem would be if that once the XML file has been updated or changed in some way, it is extremely important that any copy of this file is updated also. Otherwise this may lead to inconsistent data. This may add the extra overhead of having to update the file twice or more, and maybe even having to run another check to see if they matched exactly.

Another comment about this administration technique is that the file was 'easy to understand and edit'. This adds confidence that the program may be a step towards improving the process of module administration.

4.2.2 Test For the Students

The students test will be designed so that they need less reference to the user manual. Hopefully, it will not be necessary for a student to have the user manual when experimenting with their modules, however, this test was design to discover whether they would. The already existing method of module selection does not require any extra reading of a user manual and this would be considered a backwards step. Firstly, the student tries the test without the user manual, if they complete the tasks successfully then the interface is successful. If the student needs reference to the user manual, these problematic aspects will be considered for change.

The tasks a student must complete: (See Appendix D)

1. Input the correct prerequisites
2. Select 120 credits and submit
3. Undo Reset any selections or adhere to any warnings they receive
4. Attempt to break the rules and/or submit invalid data

The module selection software was tested on three computer science undergraduates. They were asked to imagine that they went back one year to the last time that they filled in a module selection sheet. They then had to to input the prerequisites that they had achieved at that time and proceed to select the modules that they chose last year. Further tasks were allocated such as finding information about the modules and submitting their choices. Furthermore, they were asked to attempt to break the rules on purpose and see if they could get away with it. The focus on this test was to try and collate quantitative results as well as qualitative. Quantitative to compare results and qualitative for feedback on possible program improvements. To gain figures to compare, the students were asked questions with the answer having to be a rating from one to five corresponding with bad to worse respectively. It may seem unfair testing this interface on computer science students as they have used the original selection forms so they would know how the system works. However, this has it's advantages as when they are asked to try and break the rules, they know how to fully stretch the system and can compare it to the current method of module selection.

Student's Test Results

Each of the students managed to complete the tasks. They managed to input their prerequisites and submit their selections from last year. None of them managed to break any of the rules without being warned or prevented.

The results were as follows:

To complete tasks 1 and 2 (which is all a student would require to do)
(1- Hard up to 5- Easy)
Average = 4.33
Rating = 'Quite Easy'

To try and break the rules (not necessary)
(1- 'Couldn't' to 5- Easy)
Average = 1.00
Rating = 'Could not break the rules'

How useful was the information?
(1- Not useful to 5- Useful)
Average = 3.33
Rating = 'Adequate'

How much of an improvement on the old system is this?
(1- Old System is better to 5- This is a much better system)
Average = 4.00
Rating = 'A large improvement'

How much would you prefer to use this method of module selection as oppose to the method you used last year?
Average = 4.00
Rating = 'Would probably prefer this method'

From these results, it seems that mostly, the module selection software is so far successful but with much room for improvement. These results were on the whole expected, with only the disappointment that the information supplied was only 'adequate'. Even though the information is adequate, at least there now is the option of information where as previously reference to the student handbook was necessary. The information used is taken directly from the handbook and the detail that they provide is beyond this project. It is clear to see that this software would be preferable for students over the current module selection system.

In the 'further comments' section, two of the three students mentioned the difficulties in remembering which prerequisites they had achieved in the previous year. They complained that there was no visual representation of which modules they had entered into the program also. The problem lies when, the student has finished entering the prerequisites they can remember and the main GUI starts up. They realize that on this screen there is a prerequisite they did not enter that they have achieved. A downfall in the system is that they cannot reenter more prerequisites, and have to start the program again from the beginning. It would

be simple to suggest the student sbringing along a list of their achieved modules (which should be online also).

4.2.3 Further Testing

The program was tested more thoroughly to check for general errors, mainly bugs in the program, to see which other problems may arise when wishing to perform certain tasks. The already mentioned main problem seemed to be that of not being able to add a few extra prerequisites that may have been forgotten.

Another was that the program does not terminate if the cancel button is selected on the first prerequisite request dialogue box. Not enough time was taken into account to look further into this problem but is suspected to be trivial.

The selection and deselection of modules works fine, successfully incrementing and decrementing the credits total. The reset button always resets the total to the value of the compulsory modules' credits. It also clears each of the optional modules' check boxes, ready to be selected again.

A problem found was the possibility of allowing duplicate copies of the data on the same interface. This is a major disadvantage of using this easy to use XML file as a way of storing data is that there is no check on duplicate copies of data. If there are two or more copies of a module in the file then the program will still display two copies on the interface. If the data was managed by a Database Management System (DBMS) such as SQL, duplicate copies of data would only appear once, as the only the single primary key would be used. This eliminates any ambiguities. For errors similar to this, it would be still important for a manual check to take place. This check would compare what is displayed on screen (on the interface) against the compiled list of modules and data. This would have to be checked only once before the release of the modules file. The check would be important no matter what software created the selection interface.

A Final Consideration

Currently, the program only reads a file called "modules.xml". This make this aspect of the program quite inflexible and fixed. This is fine if all of the module data is contained within this file but if different departments wish to use the software it may result in confusion to have many data files of the same name. An easy solution to this would be to have the input data file stated as a command line argument. For the the purpose of this project the simplicity of reading only one file is sufficient. This can easily be changed if the program is to be used.

4.3 Conclusion

The implementation on the whole was successful. The majority of required functions were created and placed into a graphical user interface as design intended. A 'minimal functioning' piece of software was created with a few further enhancements.

The test for the administrator was undertaken ultimately to discover or discuss the need for an administrator's tutorial. This tutorial would have been a simple walk through of each of the necessary tasks required by the administrator. On further discussion with the administrator after the test, it was clear that the stand alone XML file was sufficient with at most the aid of a user manual for reference.

The test results from the students showed that generally they would prefer this software, especially if a few of the minor problems were ironed out.

Chapter 5

Evaluation

This chapter evaluates the project as a whole and discusses if this was a step to improvement. There is an outline of how each of the goals have been achieved and a round up of the problems encountered throughout the project.

5.1 Is this program of any use?

A good way to test the success of this project is to consider if it is actually of any use and whether is it an improvement on the current existing methods of module selection or not. If the database file is easy to learn how to use then the administrators job becomes easier. It would be useful if it was available for download from the web as students may also never even have to leave their home to select and submit their modules. As the GUI has eliminated the need for paper and pencil and taken away the task of referring to the students handbook when selecting modules, a student would find use for this program.

5.2 Time Management

The schedule in general was keep to fairly accurately. The implementation of the software however, started up to two weeks before it was planned. It was realized that the implementation could be taking place in parallel with the design phase. Implementation of parts that had already been designed was necessary to take place as there were due to be many set backs whilst programming. Much time was spent learning code to achieve what was required. The anticipation of the extra time need for learning code, proved invaluable.

5.3 User Requirements Revisited

The minimum requirements for this project were:

1. Research into how modules are selected and presented

   This aspect of the project was completed successfully. Each and every possibility was considered when researching the way in which modules are selected. The current presentation of the modules was researched and requests were made to current module administrators about the history of the module system and future possible changes. This was useful during the design process as a program could be devised that would not go 'out of date' for a while at least.

2. Research into a variety of programming languages and tools

   Many of the programming languages for research were ruled out when interpreted languages were chosen over compiled languages. This led to the option of Java, Perl or Python (main stream high level interpreted languages). Java had so many advantages and a basic knowledge base of Java was already present.

3. Research methodologies concerning user interface design and Human-Computer Interaction

   There were a few methodologies considered but as the interface was only to be simple not as much research in this area was required. A few useful considerations were developed from this research and a simple methodology for user interface design was used.

4. Design a minimal functioning piece of software

   The minimal functioning piece of software was designed quite soundly and on schedule. However, extra design features were still being considered and updated until near the end of the implementation stage.

5. Implement the software

   Most aspects of the design were implemented successfully. The requirements of the specification were met. There was also some extra time allowed to implement a few of the further enhancements (discussed later). There are many more further enhancements mentioned earlier in this report which were not carried out. The program is flexible enough to allow these enhancements to be added at a later date.

6. Write a software user manual

   The user manual is complete. It covers the intended aspects and should be easy to understand and cover all possible areas of difficulty for both the administrator and student.

7. Test the product

   The test for the administrator was a success. It was designed to be fair and the results proved that the solution to the administration aspect of the project was an improvement on the current methods. The students test also resulted in a general success with room for improvement.

8. Evaluate the product

   The evaluation produced some invaluable points of discussion.

In general, the tasks were successfully completed on schedule. There were not too many serious problems with Java as a programming language and the use of XML made the objectives of this project relatively easy to achieve. Java Swing provided adequate components to create the necessary graphical user interface and were, on the whole, fairly easy to implement. Overall, each of the deadlines and minimum requirements were met as intended.

A few of the further enhancements that were covered in chapter one (The Problem) were also achieved both in design and implementation. These include the function to submit the selected results and another was to provide an easy way for the student to access further information about each module. Both of these were chosen to do with the remaining time for the project as they seemed most important. It would not have made sense for the student to only experiment with their modules and then have to write them down or still fill in one of the existing module selection forms to submit. Also, having the extra information available allows the student enough resources from simply using the software to make all decisions necessary to complete their module selection.

Another further enhancement was considered in the design phase but was never implemented. This was to show a preliminary timetable once the modules had been selected. The reason why this was never implemented is because the timetables within the school of computing are not developed completely by the time of module selection.

5.4 Problems Encountered

Finding relevant code examples from the internet and from text books created a problem which held this project up. Much time was spent searching for help,

information and sample code.

There was a problem when trying to attach a vertical scroll bar to the interface. With the use of what are called 'panels' and 'frame' within the interface, the addition of a scroll bar should have been trivial. Due to the way in which the layout had been organized, the components that were chosen, made this task more complicated than expected. Much time was spent trying to rearrange the program in order to add a scroll bar without desired success.

As the program has not been developed as an applet, this has resulted in the problem of the program's availability. Should the student be able to download it from a web page? The modules file has to be updated or different for each year. This new XML file has to be available also. One way to overcome this problem would be for the administrator to bundle all of the necessary files into one package. This could be achieved by zipping(compressing) the files into one, ready to be downloaded then unzipped (uncompressed) by the student.

Creating two 'tabbed' windows for both second and third year modules created a major problem. As this was planned to be added at a later stage in the implementation phase, difficulties arose. The components that had been chosen to create the GUI so far would have had to be changed. Many attempts were made at doing this without the required success. A simple solution if time allowed would be to have two different windows and display the third year modules in one and the second year modules in the other.

The first pop up dialogue box requesting the user's prerequisite produced problems during the testing phase of this project. Students were unable to remember which prerequisites they had already entered, and if they forgot to enter any they would have to restart the program. A solution for this would be to display visually the modules they had entered or even a list of modules they could select from. This could be placed in another tabbed window so that they could go back to it and add or remove any of the prerequisites that they had entered, without restarting the program.

5.5 If it was done again

If this project was restarted, there would be much more knowledge of Java programming and the implementation time schedule could be shortened significantly. There is no doubt that Java was a suitable language to choose for the purpose of this project and therefore, the decision to choose Java from a choice of many high level languages would not be reconsidered. This would also allow more time for de-

signing the product to avoid any of the problems that have been encountered here.

The program would be written as an applet from day one also, to overcome the problem of changing an application to an applet for use within web pages. Much of the code is reusable and many lists of code with similar command could be placed into much shorter 'loops'. The resulting code could be much shorter and more efficient. This is also important for anyone who wishes to use this code to further develop the module selection process, they would need to be able to easily understand it and quickly fathom out it's structure.

From the beginning, the interface would have a scroll bar. This would overcome the problem of having to open new windows if more data needed to be displayed. In the design phase, there would be more careful consideration of using the correct Java Swing components such as frames within panels.

There would be an earlier introduction of testing to allow further improvements before the given time ran out. There would then be time for bugs or problems that any user encountered during the testing to be eradicated.

The 'third year issue' would be considered at an earlier stage of the implementation. This would allow either tabbed windows or even extra windows. There would be a more lengthy aspect of design also regarding the prerequisite request dialogue box.

5.5.1 Further Work

As a stand alone application, this program works fine and meets its requirements. However, there are many more advancements that can be considered in order to improve its functionality. There are several possible points for improvement mentioned in chapter 1 (The Problem) 'Further Enhancements'. Obviously, there was not enough time during the life span of this project but these points may be considered by anyone who wishes to improve this software in the future.

5.6 Conclusion

Overall, this project was a success. It is a major step forward in the process of academic module selection within the School of Computing at Leeds University. The are disadvantages to the way in which the data is stored. These alternative possibilities however, are heavily outweighed by the advantages. The program can be improved up on with ease to relate to other departments within the university and even further afield to other universities. For the purpose 'to aid a students in choosing their academic modules', the choice of software as a product of research and the success of the implementation has resulted in the desired outcome.

Glossary

API (Application Protocol Interface): The interface by which an application program accesses the operating system and other services. An API is defined at source code level and provides a level of abstraction between the application and other utilities to ensure the portability of the code.

Applet: A Java program which can be distributed as an attachment in a World-Wide Web document and executed a Java-enabled web browser such as Sun's HotJava, Netscape Navigator or Microsoft Internet Explorer.

Application: A complete, self-contained program that performs a specific function directly for the user.

Check Box: An element on a user interface that allows users to select an option associated with that element.

Dialogue Box: A message from your software that pops up in a box on your screen, usually asking you a question or giving you options to choose from.

DOM (Document Object Model): A W3C (World Wide Web Consortium) specification for application program interfaces for accessing the content of HTML and XML documents. This model uses a tree structure of data.

GUI (Graphical User Interface): The use of pictures rather than just words to represent the input and output of a program. The program displays certain icons, buttons, dialogue boxes, etc. in its windows on the screen and the user controls it mainly by moving a pointer on the screen (typically controlled by a mouse) and selecting certain objects by pressing buttons on the mouse while the pointer is pointing at them.

HTML (HyperText Mark-up Language): A hypertext document format used on the World-Wide Web. Used generally to create web pages.

Hypertext: A term coined for a collection of documents (or "nodes") containing cross-references or "links" which, with the aid of an interactive browser program,

allow the reader to move easily from one document to another.

Java: A simple, object-oriented, distributed, interpreted, robust, secure, portable, multithreaded, dynamic, general-purpose programming language developed by Sun Microsystems. Java supports programming for the Internet in the form of platform-independent Java "applets".

Java Swing: An API for Java that provides a toolkit of components allowing the user to create simple and complex GUIs.

JAXP (Java Interface for XML Processing): A combination of different XML parsers designed to be used cohesively with Java. Combines many including SAX and DOM.

Module Selection Forms: (Appendix X) The forms that students in the School of Computing at Leeds University have to complete to select their academic modules for the following years.

Parsing: A method of converting data into a format that is readable by some other entity (program).

SAX (Simple API for XML): A specification for application program interfaces for accessing the content of HTML and XML documents in an event based fasion.

Tabbed Window: A component that links two panels together within one window. Allows you to toggle between different panels using 'tabs'.

XML (eXtensible Mark-up Language): An initiative from the W3C defining an "extremely simple" dialect of text suitable for use on the World-Wide Web.

These definitions are taken mostly from [9].

Bibliography

[1] M L Bernard. *Criteria For Optimal Web Design (designing for usability)*. http://psychology.wichita.edu/optimalweb/text.htm, 24/04/2003.

[2] N Bradley. *The XML Companion*. Addison-wesley Longman, 1998.

[3] A Cohn. *School of Computing Student Handbook*. Leeds University, 2002/03.

[4] A Cooper. *About Face: The Essentials of User Interface Design*. IDG Books Worldwide, Inc, 1995.

[5] H M Deitel and P J Deitel. *Java, how to program*. Prentice Hall, third edition, 1999.

[6] H M Deitel and P J Deitel. *XML How to Program*. Prentice Hall, 2001.

[7] A Dix, J Finlay, G Abowd, and R Beale. *Human-computer Interaction*. Prentice Hall, 1993.

[8] E R Harold. *Processing XML with Java*. Adison-Wesley, 2001.

[9] L Johnson. *Free Online Dictionary of Computing*. http://wombat.doc.ic.ac.uk/foldoc/, 14/03/2003.

[10] B Milewski. *The battle of Languages - Java vs C++*. http://www.relisoft.com/Web/c-java.html, 10/12/2002.

[11] P Niemeyer and J Knudsen. *Learning Java*. O'Reilly & Associates, second edition, 2002.

[12] A Smith. *Human Computer Factors*. McGraw-Hill, 1997.

[13] Inc Sun Microsystems. *The source for java technology*. http://java.sun.com, 10/12/2002.

Appendix A - Reflection on the Experience

This project has given me much insight into software development and time management. It has given me the opportunity to develop my programming skills not only in Java alone but many techniques and methods of programming have been learnt. With experience of encountering so many problems within the code itself, and towards the original design of the product, my intuition has increased and knowledge widened.

I have discovered that sound design from the beginning of a project would result in fewer problems in the long term. Trying to add extra functionality as an afterthought may not fit in with the original design causing further problems.

Many aspects of module administration have been discussed during this project and I have increased my awareness of how university departments handle their information and data to do with module enrolment.

There was a constant check on the schedule and time management became a large part of the project. Being able to alter the schedule and extend other aspects of the project was a careful task. It was very important for a diary to be kept of the work I had done and what I had discovered, it helped me greatly when I came to write this report.

I have learnt about the differences between various high level languages. The difference between interpreted and compiled languages and what many of them are capable of with respect to application development. If I am required to produce an application in the future, I can be wise with my choice of programming language.

This project resulted in my first encounter with XML files. I have found that XML files can be extremely versatile and can very useful under that right circumstances. However, the disadvantages of this method of data storage have also been noted. Comparisons were made between other methods of data storage (DBMS and text files) to help me realize this.

I have developed my skills in report writing and with a fair amount of time spent learning Latex (aids with the format of this report), I shall be more efficient the next time I write a report.

Finally, I have learnt the importance of asking for help when needed. The regular meetings with my supervisor helped largely with the development of my ideas and programming techniques.

Appendix B - Module Selection Sheets

Appendix C - User Manual

Appendix D - Testing Material