PROJECT SUMMARY

Ioannis Stavrakas
MSc in Computer Science course
Department of Computer Science
University College London
Summer 1992

Supervisor: Nigel Chapman

DESIGN AND IMPLEMENTATION OF AN OBJECT ORIENTED BIBLIOGRAPHICAL DATABASE

A bibliographical database keeps details about items that can be accommodated in a library. This project has been originated by the need for a program for the research library of the Computer Science Department. The main aim was the design and implementation of building blocks, that can be used for a quick development of a program offering functionality tailored to specific needs. A secondary aim was the development of a user program example, that demonstrates how the building blocks can be used.

We used the C++ language and object oriented techniques, to design and implement both the building blocks and the user program. The final result is a database that can keep information about nine different kinds of bibliographical items: books, periodical articles, monographs, forthcoming books, forthcoming articles, departmental research, theses, microfilms, and computer software. The items are classified to four categories: published, forthcoming, unpublished, and non-paper. The program offers facilities to update the database, retrieve items that satisfy certain conditions, and navigate through the different levels of the database. Saving and reading from files is transparent to the user.

Every effort has been made to facilitate future extension of the functionality provided by the building blocks. The greatest reward for this effort would be to be found useful.

DESIGN AND IMPLEMENTATION OF AN OBJECT ORIENTED BIBLIOGRAPHICAL DATABASE

Ioannis Stavrakas
MSc in Computer Science course
Department of Computer Science
University College London
Summer 1992

Supervisor: Nigel Chapman

A bibliographical database keeps details about items that can be accommodated in a library. This project has been originated by the need for a program for the research library of the Computer Science Department. The main aim was the design and implementation of building blocks, that can be used for a quick development of a program offering functionality tailored to specific needs. A secondary aim was the development of a user program example, that demonstrates how the building blocks can be used.

We used the C++ language and object oriented techniques, to design and implement both the building blocks and the user program. The final result is a database that can keep information about nine different kinds of bibliographical items: books, periodical articles, monographs, forthcoming books, forthcoming articles, departmental research, theses, microfilms, and computer software. The items are classified to four categories: published, forthcoming, unpublished, and non-paper. The program offers facilities to update the database, retrieve items that satisfy certain conditions, and navigate through the different levels of the database. Saving and reading from files is transparent to the user.

Every effort has been made to facilitate future extension of the functionality provided by the building blocks. The greatest reward for this effort would be to be found useful.

CONTENTS

1. INTRODUCTION

The application of computer science in libraries is a good example of how useful a computer program can be. Useful for both the users and the administrators of the library, and in more than one way: finding whether a specific book exists in the library and its location, is just the beginning; sorting and retrieving information about items that satisfy complex query conditions are fields where computers do well.

Of course, you must keep in mind that there is always a price to pay for shortcuts: often you learn in the process of trying to find something yourself; the computer will give you instantly the answer, minimising the time, but depriving you of this knowledge and of the pleasure of browsing on shelves.

1.1. Overview of the project

This project is about a program for a special kind of library: a research library.

1.1.1. The need

A research library can differ from a general library in the following aspects:

a) the material in the research library refers to a more specific field.

b) The number of the items in the library is usually smaller.

c) the users of the library have different demands (for example, they often do not look for a specific item, but for items that refer to a subject of interest).

d) in a research library it becomes important to keep items that are not normally kept in libraries, such as PhD theses, departmental research, and since we are talking about the research library of the Computer Science Department, information about computer software as well. What is more, the distinction between published, forthcoming, unpublished and non-paper (broadcasts, interviews, software etc.) items, becomes important.

The above reasons suggest that finding the location of an item in such a library must not be the primary concern of a computer program. On the other hand, it becomes very important to keep information about the structure of the library (division to published, forthcoming, unpublished, non-paper items) and about the contents of each item. Useful operations could include listing and counting all items that are of the same type (for example: count all unpublished items), retrieving all items that refer to or mention a combination of keywords, creating footnotes or lists of bibliographies (correctly formatted and sorted) in a document.

1.1.2. Existing programs : what they do - what they do not do

Unix has a collection of programs[1] to create and update a bibliographical database. It also provides a basic retrieval facility and capability of creating footnote and bibliography references in an nroff or troff document. Tib[2] is a similar collection of programs for TEX documents.

They both include programs that :

a) create inverted indexes for the databases

b) create footnotes or bibliography lists

c) retrieve references that match a combination of keywords

- Unix collection provides some simple sorting and formatting facilities, plus a simple interface to update the dbases.

- Tib offers more sophisticated options, with many different predefined formatting styles for bibliographies, including abbreviations etc.

However, these programs do not completely cover the needs of a research library because of their limitations :

a) they cannot perform operations on all items belonging to the same type (ie. count all PhD theses, list all forthcoming items, retrieve only the articles that satisfy a given condition).

b) they do not know which attributes are associated with each type of items, so they cannot prompt you to give the information associated with a specific item when updating the dbase.

c) you cannot associate keywords (which will refer to content or subject) to different chapters or page numbers of an item. If you could do this, the program would

be able to maintain an "index" for every item, much like the indexes at the end of books, which would permit queries based on the content of the items and replies giving page numbers as well as book titles and authors.

1.1.3. A different approach

The above discussion, suggests the use of objects and relations between them as a means to model the database, each having some particular attributes and operations associated with it. We observe that the relational model does not provide a natural way to capture these relationships, and that this application does not need "join" operations, which is the strong point of a relational DBMS. These lead us to the adoption of the object oriented model for our dbase.

An advantage of this approach is that it solves the previously mentioned inadequacies of the existing programs: the structure of the dbase is now explicitly defined and operations can be implemented on this structure, each type of items "knows" which attributes correspond to it, and it is possible to maintain an "index" of contents and pages for each item.

An additional advantage is versatility: it becomes easier to create a different item type when you need it, or add functionality, without starting from scratch, by using the already available pieces of code.

However, it is useful to recognise early that this model may suffer from a classical problem of hierarchical structures: it may be difficult to implement efficient operations, that involve parts of the structure that have different parents. This kind of operations, therefore, must be treated with special care during the design.

1.1.4. Getting started

The first step is to understand the real-world part of the problem, in our case to identify the types of items that the library contains and the attributes of each type of items. By carefully studying the Tib manual[2] and the "Manual for Writers of Research Papers, Theses and Dissertations" by Kate L. Turabian[3], we can built a model of the real world entities (see APPENDIX B) that contains all the information needed to make bibliographical or footnote references to items.

The next step is to understand what the problem involves, by carrying out a very high level functional analysis, that will state operations on our dbase model that may be useful to the users of the library.

At this stage, one understands that it is not possible to implement all this functionality in the three months that the project lasts, so there must be a compromise.

1.1.5. The project

There are three possible approaches to this compromise :

a) choose some of the dbase item types and a part of the possible dbase operations, and design, implement and test completely a system - which however will be incomplete and insufficient as far as the real world is concerned.

b) design, implement and test the object types (item types), but do not get involved with high level functionality (how the objects will become useful to the users of the library).

c) design, implement and test some object types. Then design and implement an incomplete program, that uses the objects to demonstrate how useful operations can be performed by using the member functions of the objects.

This project will adopt the third approach. The emphasis will be on the design, implementation and testing of the building blocks, that can be used for the quick development of programs with functionality tailored to different needs. Reusability is, therefore, the first priority. This means that we must anticipate what users might need, but also that the classes must not be too helpful, but leave the decision making to their users, where possible ("Perfection is achieved, not when there is nothing left to add, but when there is nothing left to take away" - St Exupery).

A program that uses the classes to provide some high level operations on the dbase will also be produced. It is not intended to be robust, complete, or thoroughly tested, but just to demonstrate that the classes work, and how they can be used. However, the final program is considered to be a vital part of the project, that will be implemented even at the expense of some of the objects (an initial time schedule and the actual time spent on the activities involved are included in APPENDIX A).

So, at the end of the project we expect to have a complete design and code for the objects of the system, plus a short design and code of a program that demonstrates how the classes work.

1.2. Definition of the problem

Before starting to think about the solution of a problem, we need to define exactly what this problem is. First, we need to agree upon what we are going to solve, and then worry about how to solve it. In order to understand the problem, we need to learn about and become familiar with the real world entities involved. In our case, we need to model the library, its structure, the objects that belong to it and their attributes. Then we can sketch an outline of functional requirements to which our solution must conform.

1.2.1. Model of the library

The model of the library can be divided in two parts: The first part recognises the objects that form the library and the relations between them. We use an Entity - Relationship Diagram to show the structure of the library. The second part defines the attributes for each item.

It must be emphasized that this model is not a part of the design of the system, although there is a relation between the model and the design.

Structure of the library

The E-R Diagram (figure 1.1) starts with a conceptual division of the bibliographical items to published, forthcoming, unpublished and non-paper. The items of each of these types may then consist of other items (ie. a multivolume work consists of volumes, a newspaper of articles etc.); this division corresponds more to a physical division than to a conceptual one. Useful information about the physical structure of the bibliographical items can be found in the "Manual for Writers of Research Papers, Theses and Dissertations"[3] and the Tib manual[2].

Some of the items mentioned in the former book have not been included in the E-R Diagram, because they were found irrelevant to a computer science research library. These are: plays and long poems, medieval works, scriptural references, and classical references. Series have not been included because they consist of volumes which can be considered as books without loss of information. Encyclopaedias and dictionaries have not been included as it is impractical to enter details for every article they contain, and as they are similar to collections. Novels however, have been

Fig. 1.1 : E - R DIAGRAM

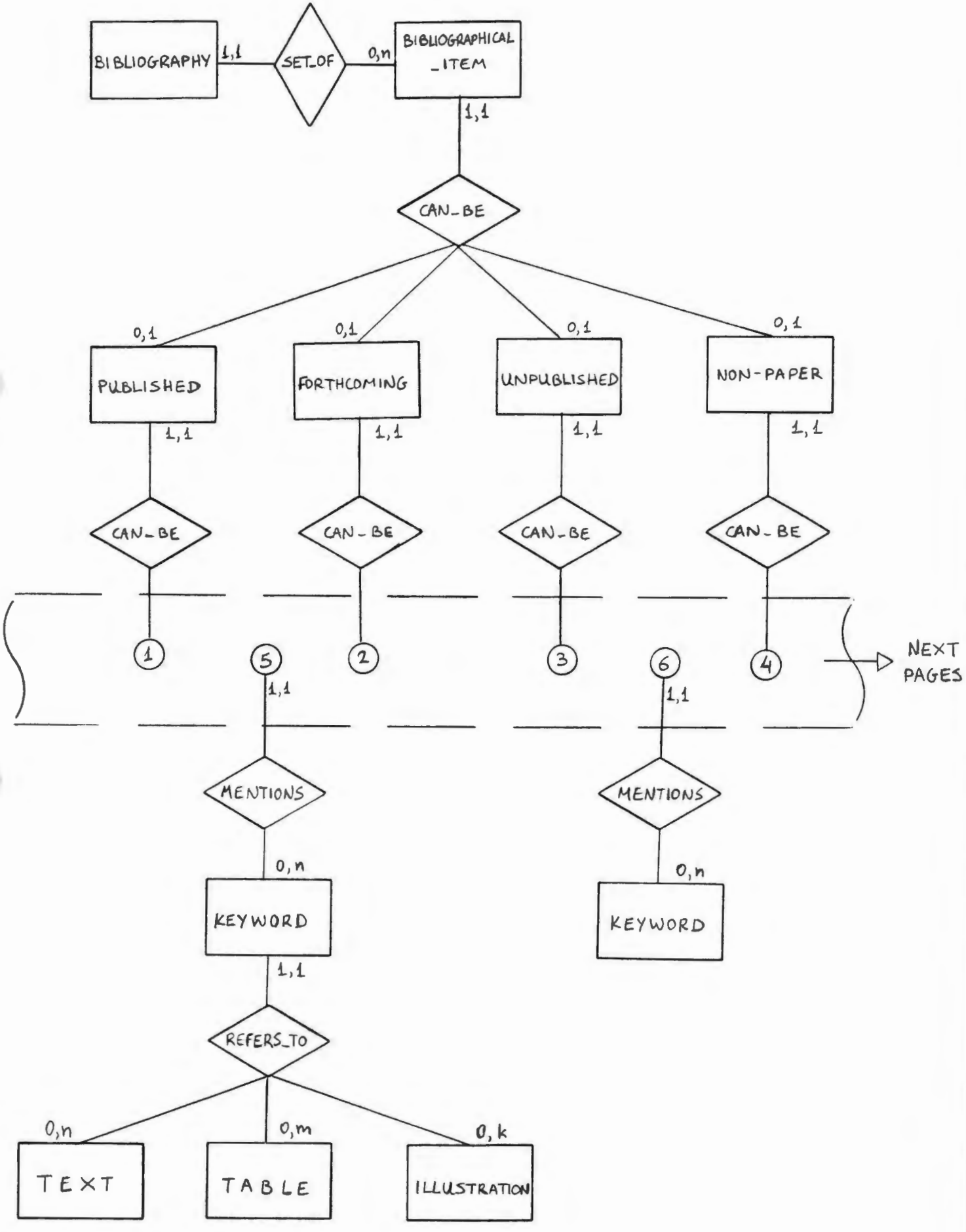
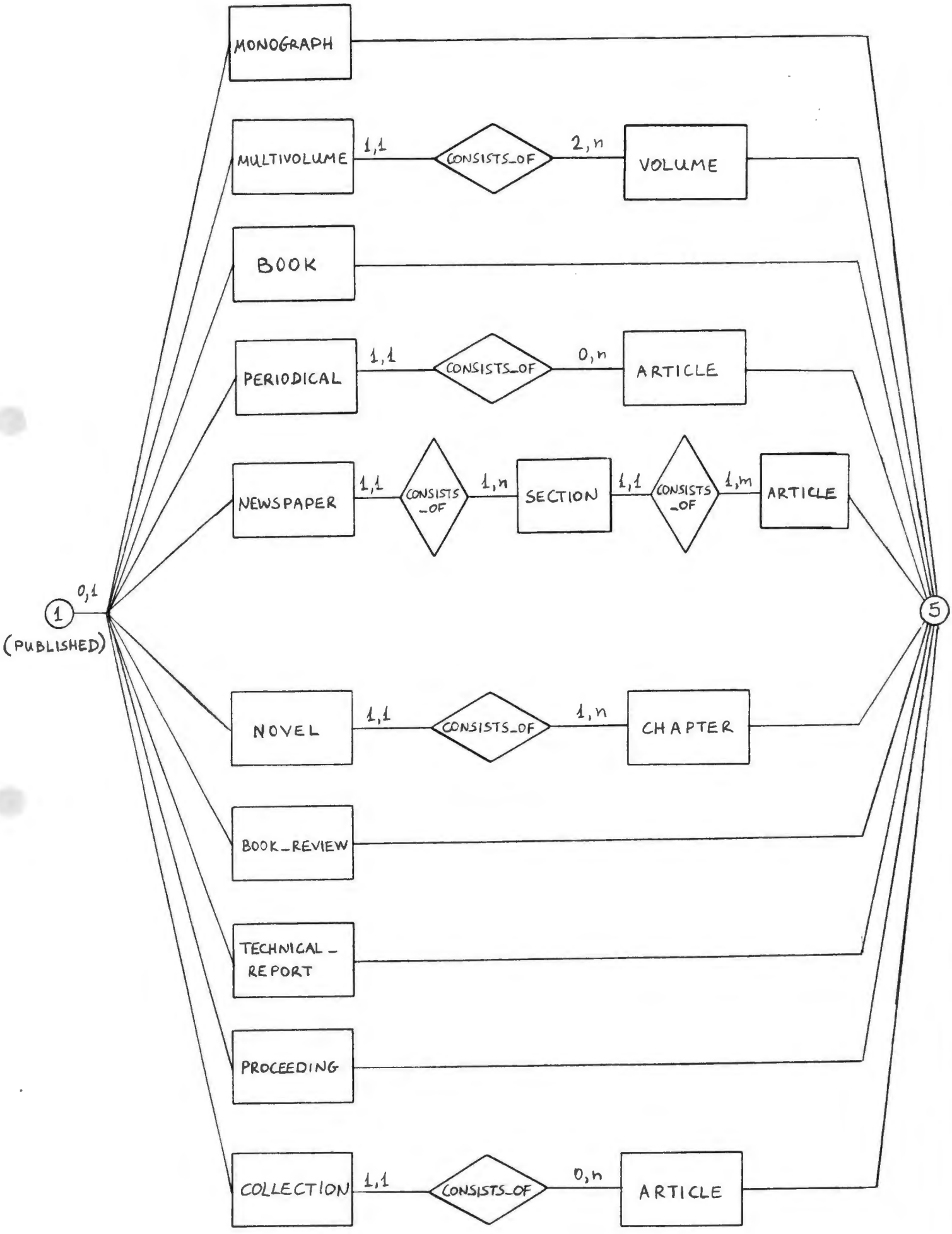


Fig 1.1 : continue



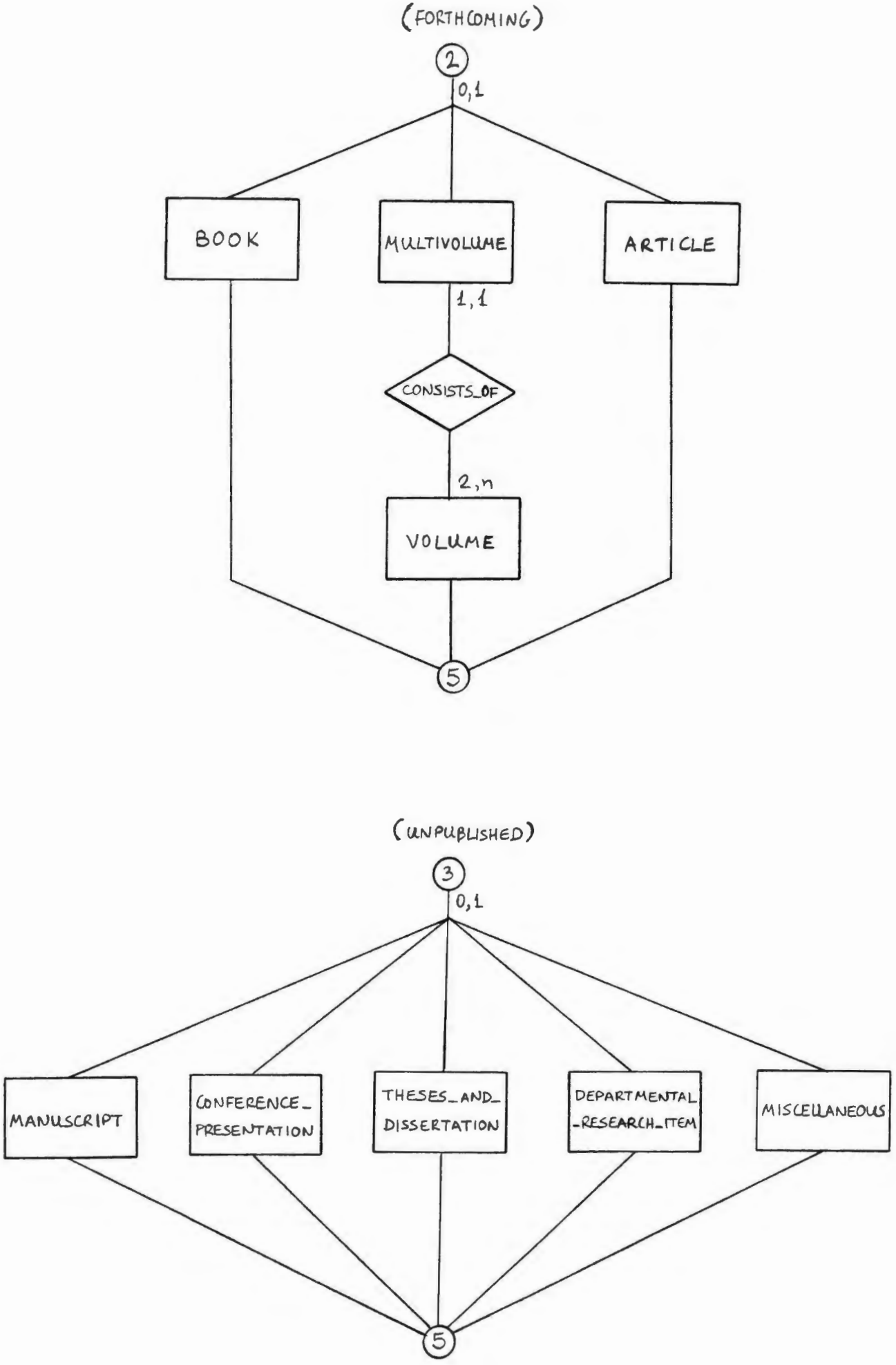


Fig 1.1 : continue

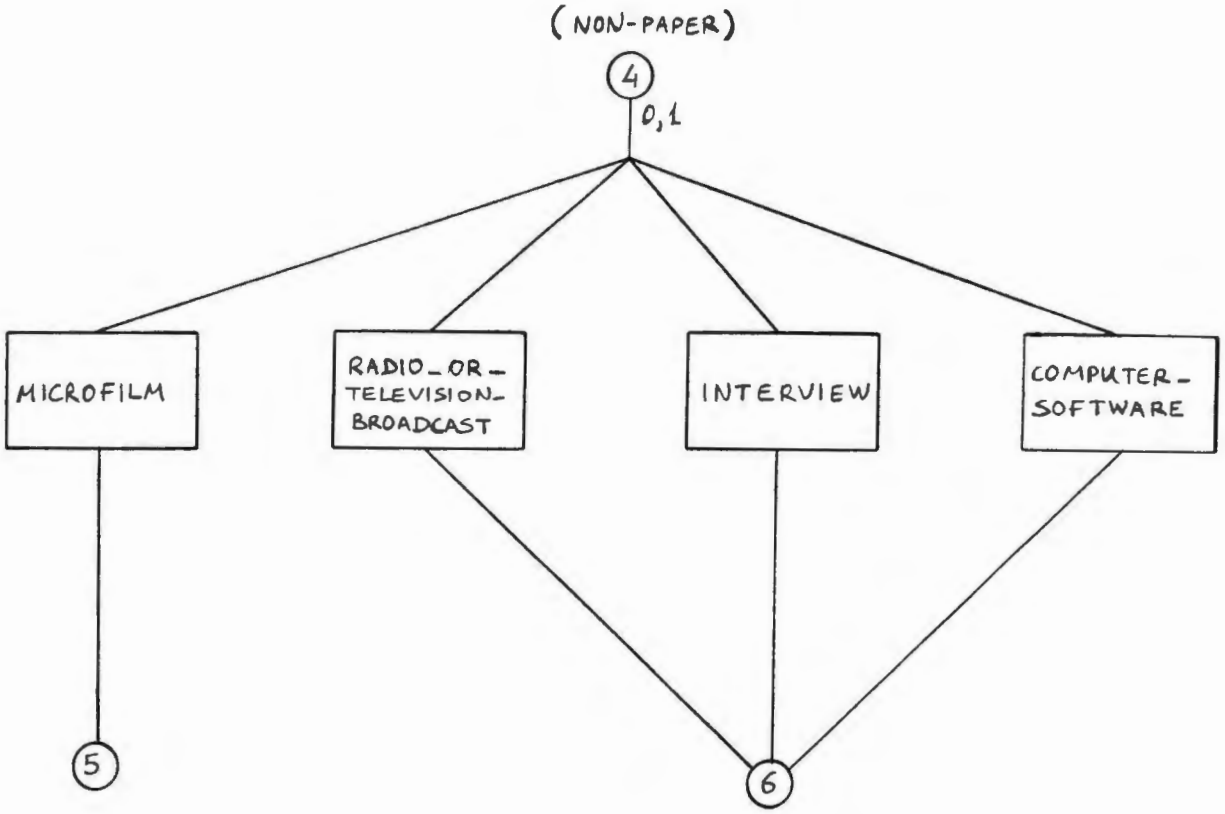


Fig. 1.1 : continue

included.

In future more objects might be included. For example, a relatively new kind of published items is the CD ROM which does not belong to any of the above categories. Electronic publishing is another recent way of publishing which could be added to our database structure.

The E-R Diagram then concludes with another conceptual division related to the design: the keywords that refer to the content of an item and can be used for retrieval of items. If the item does not belong to non-paper items, the keywords can refer to tables, illustrations, or simply text, and have page number(s) associated with them.

Finding the attributes

The listing of the attributes considers only the items that are physically part of the library. For example, the attributes of a book have been defined, but not the attributes of a published item in general. The attributes of the items have been selected by studying the "Manual for Writers of Research Papers, Theses and Dissertations"[3] and the Tib manual[2].

As an example, the attributes of the "periodical article" are given here. The complete lists of attributes of all the objects that exist in the E-R Diagram and are physically part of the library, are given in APPENDIX B.

PUBLISHED ITEMS: PERIODICAL ARTICLE

- PERIODICAL :title - volume number - publication date - number of pages
  ARTICLE :authors (including pseudonyms) - title - page numbers

NOTE : "number of pages" refers to the total number of pages of an item, while "page numbers" means the numbers of the first and last page of a part of an item.

1.2.2. Functional requirements

The facilities that our final program aims to offer can be divided into four kinds :

1) Operations on the structure of the dbase.

These will permit the user to navigate through the structure of the dbase to different levels, in a way similar to the way that we move from one directory to another. These levels will represent sets of items, like set of forthcoming items or set of books, and specific item instances. The user view of the system is shown in figure 1.2. This approach has considerable advantages :

a. the available operations can depend on the current level. For example, on the level UNPUBLISHED ITEMS you could have the operation "count" and on the level NEWSPAPER you could have "list articles" of a particular newspaper.

b. the same commands will be able to do different things depending on the level. For example, the command "count" will count the number of books if you are on the level BOOKS or will count the number of forthcoming items if you are on the level FORTHCOMING.

This feature could be applied on facilities that create footnotes or bibliographies : together with the keywords, you could specify the level on which you want the searching to take place, restricting the operation (example : search only for PhD theses that match the keywords).

c. navigation will provide an elegant way to solve the update problem, which is described below.

2) Update operations

We only consider the ADD operation here, but operations to modify and delete items will suffer the same problem; there must be two levels of update operations:

2.1 One that will add new items (periodical, book etc) to the dbase, prompting for the information associated to each item.

2.2 A second that will add information to an already existing item. For example: add a new keyword for reference, associated to page number(s), to the index of a specific book.

This problem could be solved by associating the different kinds of update operations to the different levels of the dbase structure as described above. There could be

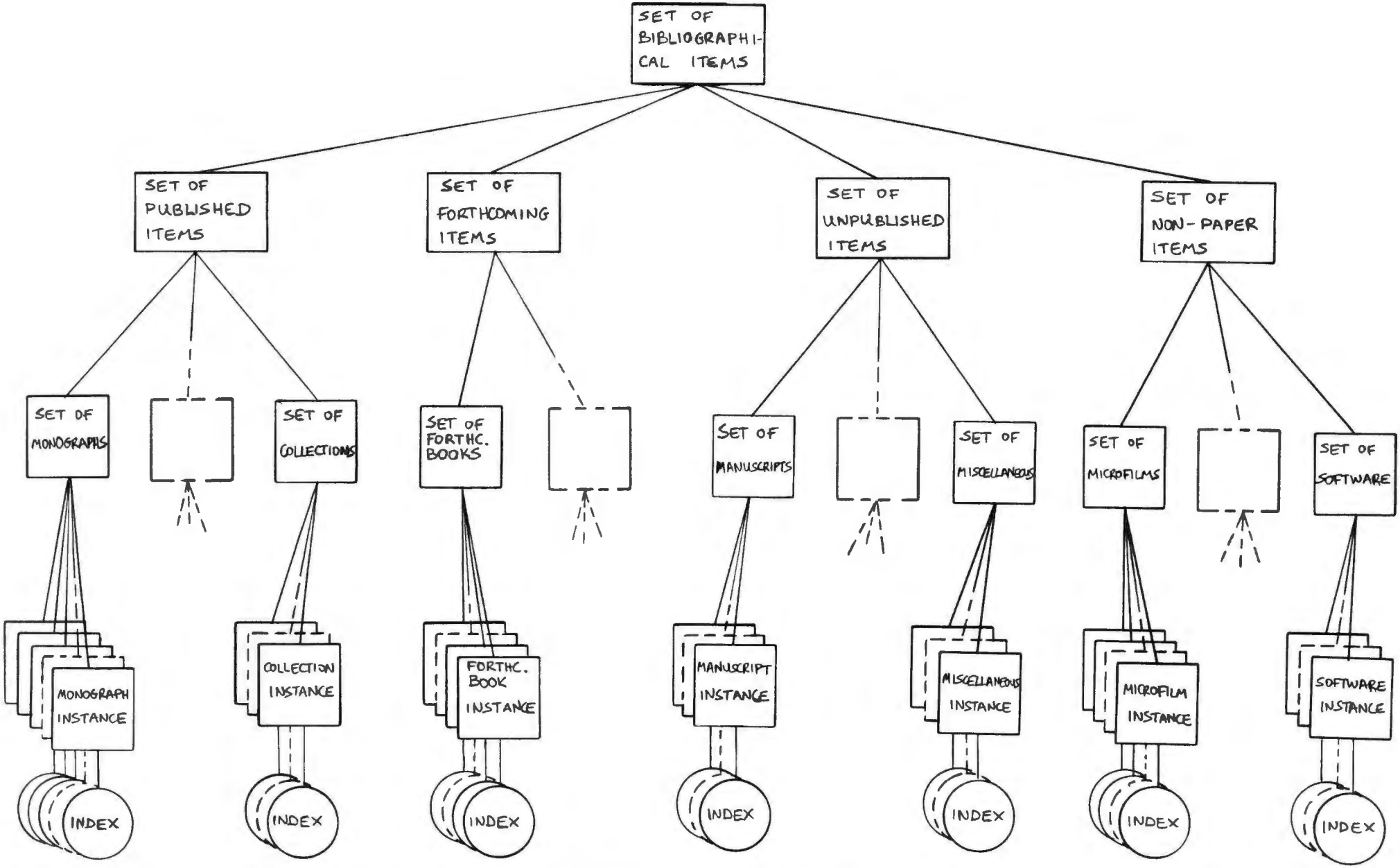


Fig. 1.2 : USER VIEW OF THE SYSTEM

a sub-level corresponding to the index of an item instance, that would permit the user to build an index of keywords that refer to the content of an item and are associated to page numbers, indicating where a particular keyword is encountered in the item.

3) Retrieve operations

3.1 Give a keyword and see all the items of the current level that match the keyword. The keyword is matched with some attributes of the item and need only be part of an attribute value, not the complete word. Which attributes will be matched, will depend on the kind of item, as different kinds have different attributes with varying importance.

3.2 Give a keyword and see all the items of the current level that contain the keyword in their index. The keyword must be the complete word.

These operations will not be affected by whether the keyword will be in capital or lower case letters.

4) Auxiliary operations

Except from the above operations, the classes will provide functions that will enable a program that uses them, to offer facilities like :

- move items from unpublished to forthcoming and from forthcoming to published

- see the number of items of a given type (books, non-paper, etc).

- list all items that belong to the same type (example : list all unpublished items)

- see the "index" of an item (the subjects it refers to), after having specified the item

- find the page of a specific item that refers to a keyword, if the keyword exists in the "index" of the item.

5) Operations to be added in future

The design of the classes must also permit them to be extended, so as to be able to provide operations like:

- create footnotes or bibliography references, by giving keywords and specifying the level of the dbase where you want the operation to be carried out (explained above).

- sorting and formatting facilities.

2. DESIGN

Top-down? Bottom-up? Inside-out?

Everything would be so simpler, if we could choose one of the above methods and design a system according to it. The problem is that we have been able to use these methods not for designing the system, but for presenting the design. The actual design activity has been a mental process very similar to arranging the small pieces of a puzzle so that they form a picture. You have to take a close look, examine the details, then go back, see if they fit in the big picture, and repeat the same for as many times as necessary. It would be inefficient to constrain the way you think by imposing a top-down or bottom-up approach.

Maybe this happened because we did not have a specific target to reach, as it would be the case if a customer existed. We had to anticipate how the classes might be used in future by different user programs, and design them as flexible as possible. The problem here is that, when you have a doubt you cannot ask the customer, because there is not one, you have to make all the decisions yourself, and finally you have to compromise somewhere between the ideal product and the reality of the time you have available.

For the rest of the report we assume that the reader is familiar with the C++ language[6] and the concepts of object oriented programming[9,10].

2.1. Introduction to design

According to D. Ince[4] the basic activities are:

1. Define the relationships between the design entities, and outline the structure of the system.

2. Determine the interface of the classes that represent the design entities.

3. Outline the tests that will check whether the functions of the classes do what they are expected to do. The tests will be finalised during the implementation.

4. Select the data structures that will be used to implement the abstract data types.

However, before starting to think about the structure of the system, we need to identify the entities of the real world that are going to be represented by classes in the system. Not all the real world entities will become system entities, and the relationships between real world entities will not necessarily remain the same between the corresponding system entities.

The real world model described in section 1.2 of the previous chapter has some points that need modification in order for the design to be simple and efficient, without sacrificing its functionality:

- In the real world a volume is part of a multivolume work. However, since each volume may refer to a different - although not completely different - subject, we think that the volume is the conceptual unit that must be emphasised in the design, and the multivolume details could be attributes of the volume. This makes sense if we make the assumption that we are more interested in operations on a volume (like find volumes that refer to a subject) than in operations on multivolume work (like find the titles of the volumes that belong to a multivolume work).

- For the same reasons, articles are considered more important than the periodicals and the newspapers that they belong to, and are treated separately with details of periodicals or newspapers as their attributes. The same holds for collections of articles.

There is an obvious doublication of information, since we repeat some attribute values for many article or volume instances, but when selecting the entities for the design, it seems better to think about different entities as objects with conceptual integrity that talk about specific things, than just as objects that are physically independent and are not part of other objects.

The last approach would permit operations like: list all articles in a specific newspaper. We think however that this would not be a very useful operation, especially if the newspaper is available, as it might be simpler to check the newspaper itself. Maybe it would be more useful to concentrate on the efficiency of operations like: find all articles that talk about a specific subject. In this case it is not very simple to check all the newspapers one by one.

2.2. Structure of the system and design of the classes

First we will explain the role and the relations between the objects that form the system, and then we will outline the operations that each of these objects will be able to perform.

2.2.1. Structure of the system

Figure 2.1 shows the relations between the objects that form the system. The heart of the system is a collection of sets of various kinds of bibliographical items (set of books, set of newspaper articles, set of microfilms, etc). Each set is an instance of a polymorphic set [33], that contains items of the same kind [11-32] (books, novels, microfilms etc), and may be associated with a file where the items are kept. The polymorphic set will offer facilities for retrieving items by giving a keyword that corresponds to some attributes or by giving a keyword refering to the contents of the item, adding and deleting an item from the set, counting and listing the items in the set.

The polymorphic set will be defined to contain abstract bibliographical items [10] with virtual function declarations: compare an item to another, prompt for the "key" attributes that specify an item etc. Each type of items will inherit this abstract class and define the virtual functions according to its specific attributes.

Some additional virtual functions declared in the bibliographical item class will be: print an item, save or read an item from disk, modify an item, prompt for the attribute values, retrieve the attributes of the item.

Each item will also include an associative array [9] that will be used as an index to store keywords refering to the contents of the item, and page number(s) associated to the keywords that will show where in the item information about a keyword can be found. The associative array will offer operations to check whether a keyword exists and the associated page(s), list all the keywords, add and delete keywords and page number(s).

The abstract bibliography item is inherited by all classes representing the different types of items [11-32]. Each type defines the virtual functions that need redefinition and adds any other functions specific to that type (for example some items are likely to be moved from unpublished to forthcoming or from forthcoming to

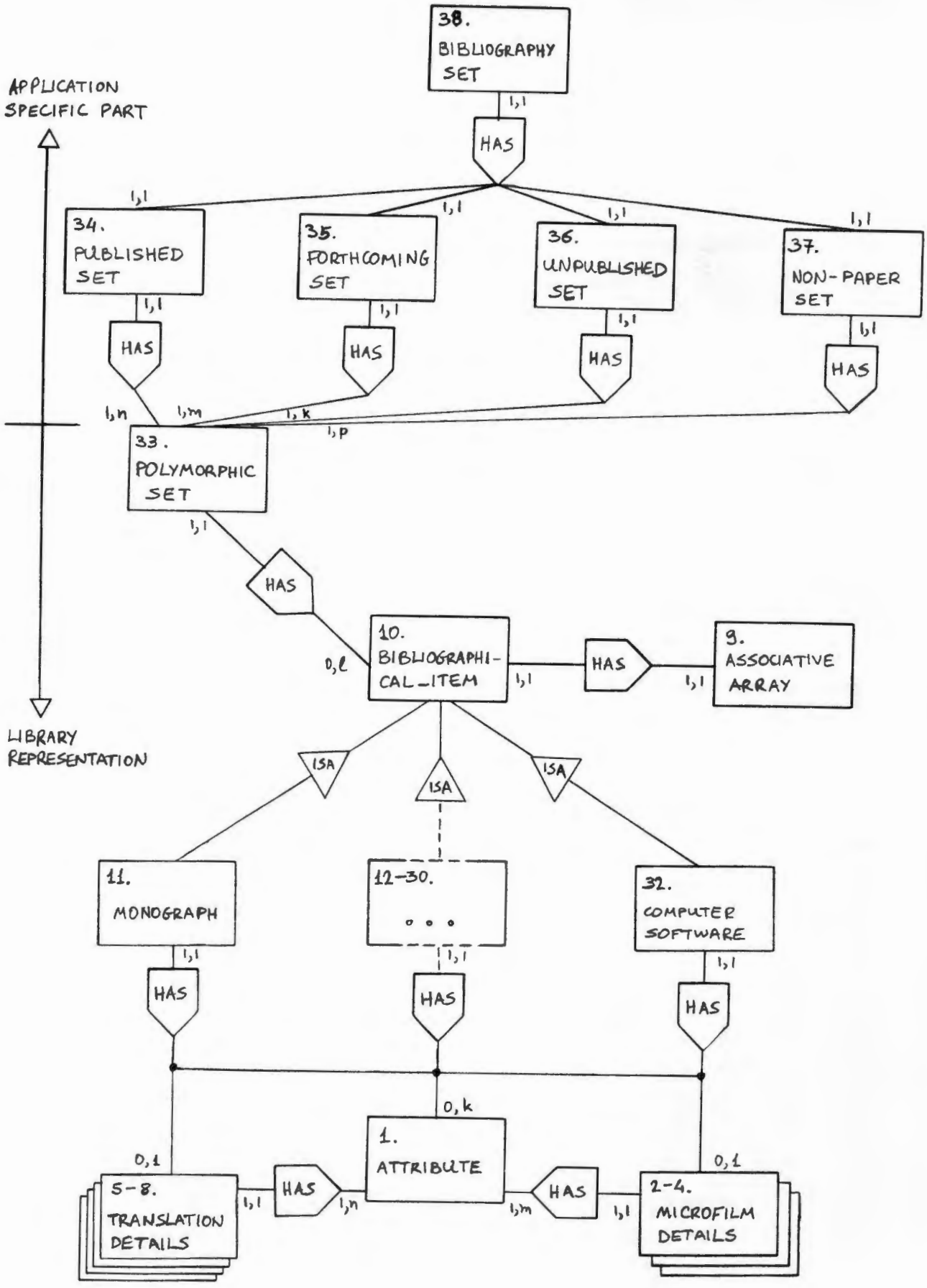


Fig. 2.1 : STRUCTURE OF THE SYSTEM

published). It also declares its attributes as instances of the Attribute [1] class.

The Attribute class contains operations that store and return the value of the attribute, save and read an attribute instance from disk, modify and print the attribute to the standard output, and check whether the value of the attribute is the null string.

The types that can be translated contain a pointer to another class similar to classes [11-32] that keep information about the original item [5-8]. The microfilm type contains pointers to classes representing all the different things a microfilm can hold details for. In our case, a microfilm can hold details about a book, a periodical article, or a newspaper article. This is convenient since we can use pointers to the already defined classes of those items, inside the microfilm class, and we do not need to define additional classes to hold the microfilm details.

The sets of bibliographical items can be grouped together to permit operations that act on many of them at the same time. Each of the published [34], forthcoming [35], unpublished [36], and non-paper [37] set classes will use the corresponding instances of the sets to provide operations such as count, list items and search for a specific item, on the union of the sets that they represent. The bibliography set class [38] will use the previous classes to provide similar operations on all the stored items.

2.2.2. Design of the classes

In our opinion, this has been, together with the design of the structure of the system, the most difficult part of the project, since most of the important decisions had to be taken here. Decisions that affect not only the implementation, but the view that the user programs have of the underlying system (the classes), the degree to which it will be possible to extend the classes so as to provide more facilities to user programs with different requirements, and the possibility of improving the efficiency of the system without changing the interface of the classes - without changing the user programs, that is.

It is good to be tidy, so the first thing to consider is some rules about storage allocation. There will be a clear separation of responsibilities between the classes and the client program, about the storage allocation: each will be responsible of deleting the storage that it has allocated. When the classes need a structure created by the client program, they will always make a copy of it, and the client program will be responsible for deleting the structure that it has created.

However, there is an exception to this rule: some member functions of the classes will create new objects and pass pointers to these objects to the client program. For example, when you specify an item of the set in order to modify it, the member function of the set that finds the item will make a copy of it and return a pointer to the copy. In this way the set is protected from implicit modifications. The client program must know which the functions that return new copies are, and delete the structures when it is appropriate.

When determining the interface of the classes, it is a good idea to push the operations lower in the hierarchy, where this is possible. For example, by looking at figure 2.1, we see that an instance of a polymorphic set contains one kind of bibliographical items, for example monographs. Monographs contain a number of attribute instances and an associative array (index). When we decide that we need a "save" and a "read" function to write and read the contents of the polymorphic set from a file, we imply that the monograph class must have similar functions operating on one monograph instance, so that the set functions will use the monograph functions for all the monograph instances in the set. This also means that the associative array and the attribute class must have similar functions as well, that will be used by the corresponding functions of the monograph class. In this way, the functions are shorter, easier to modify, clearer to understand, and easier to test.

A short description of the classes follows, in the order that they have been implemented and tested:

[1] Attribute

There are functions to set the value by making a copy of a given string, and check the value that will return a pointer to the stored string. These are the most general and flexible functions and actually the only ones that are absolutely necessary. However as we already explained, more functions at this level will simplify the work we will have to do for the rest of the classes. So, there are functions that check whether the value is the null string, functions that read and write the value to a file (they receive a pointer to a stream connected to a file), and finally additional functions that are associated with the standard input - output and get the value, print, or enable one to modify it.

[9] Associative array

The "put" function inserts a pair of key-value if the key does not already exist, or else it sets the value of the existing key to the new value. The value associated with a key can be checked by the function "at" that receives a key and returns a pointer to a value, or 0 if the key does not exist in the array. There are also functions to remove a pair key-value from the array, to count the existing pairs, to save and read the associative array from a file by specifying a stream, and to print the index in the standard output: keys on one column, values on another. Another function returns a pointer to an exact copy of the associative array. This will be useful later on, when we will need to make copies of bibliographical items, since a bibliographical item contains an associative array.

[10] Bibliog_item

The public part includes an Assoc_array [9] declaration, a pointer to Bibliog_item which will be used to link item instances together in a list, plus the following function declarations, which are declared as virtual and are not defined in this class, but each item type inherits them and defines them according to its own attributes.

compare: receives a pointer to a Bibliog_item and returns 0 if the two items have the same key attribute values, 1 if the current item is greater, and -1 if the item pointed by the parameter is greater. All the key values are used for the comparison, but some are more important than others. For example, the titles of the two items will be compared only if the names of the authors are the same.

compare_attribute: receives a pointer to a keyword and returns 1 if the keyword is part of a specific subset of the attribute values (ie author, title), else returns 0. The keyword need not be a complete word and can be given in small or capital letters. The attributes that are matched depend on the kind of the item (not all kinds of items have authors or titles).

print_attribute: receives a pointer to ostream and lists the attribute names and values of the item. The stream may be the standard output or may be associated to a file.

copy_item: creates an item of the same type, copies the current item values to the new item, including the attribute values and the associative array, and returns a pointer to the new copy. This function will be very useful to the polymorphic set class when inserting a new item, or specifying an existing item.

The following two functions are used by the polymorphic set to save or read all the items of the set from the disk:

save_item: receives a pointer to ostream and saves the item to the file connected to the stream.

read_item: receives a pointer to istream and saves the item to the file connected to the stream.

The rest of the functions work with the standard input-output:

prompt_for_key: prompts for the attribute values that consist the "key" of the item and saves their values in the item

prompt: prompts for the various attributes of the item type, and stores the values in the item.

modify: shows the current values of the attributes, prompts for new, and stores then in the item

print_key: prints the names and values of the key attributes of the item to the standard output.

[11-32] Item types

These classes inherit the Bibliog_item class, define the virtual functions according to their specific attributes, and declare various Attribute instances as private data. Item types that may have authors, compilers or editors, will include a private variable denoting whether the relevant attributes refer to authors, compilers or editors. The item types that can be translations (Volume, Book, Novel, Technical_report) and the

microfilm class must be treated with care because of their varying number of attributes: they will keep in their private part a pointer to a class holding details about the microfilm type or the original item of the translation. Microfilm instances will contain three pointers to the classes that describe the nature of microfilm (book, newspaper article, periodical article); one of them will actually point to the relevant class instance, the rest of the pointers will be 0.

The classes are:

[11] Monograph

[12] Volume

[13] Book

[14] Periodical article

[15] Newspaper article

[16] Book review

[17] Novel

[18] Technical report

[19] Proceeding

[20] Collection article

[21] Forthcoming volume

[22] Forthcoming book

[23] Forthcoming article

[24] Manuscript

[25] Theses

[26] Conference presentation

[27] Miscellaneous

[28] Research

[29] Microfilm

[30] Broadcast

[31] Interview

[32] Software

Some classes whose instances can be transfered from unpublished to forthcoming or forthcoming to published, will include functions names similar to:

make_forthcoming: creates the corresponding forthcoming item, copies all relevant info to the new item, prompts for the non-common attributes of the two item types, and returns a pointer to the new item.

make_published: creates the corresponding published item, copies all relevant info to the new item, prompts for the non-common attributes, and returns a pointer to the new item.

[2-4] Microfilm details

A microfilm can be about a book, a periodical article, or a newspaper article. We already have classes about these items, so the microfilm class will contain in the private part pointers to all of these classes plus the attributes that are specific to microfilms. Two of the three pointers will always be 0 and only one will point to a class instance, depending on the content of the microfilm instance.

[5-8] Translation details

These are classes that hold details about the original when the item is translated. Except from the appropriate attributes, they have functions like "prompt", "modify", "print_attribute", "copy_item", "save" and "read", similar to the corresponding functions of the item types. The items that can be translated will contain a pointer to the relevant translation details class in their private part, and use the functions listed above in their own similar functions.

[33] Polyset (polymorphic set)

This can be initialised with a pointer to a filename as a parameter to the constructor. The destructor will save all the items overwriting the specified file or creating it if it does not already exist (uses the save_item of each item). However a set instance can

be created without a filename and in this case, it is not connected to a file.

add: receives a pointer to a Bibliog_item, and checks whether it already exists in the set (uses the compare function of the item). If it exists 0 is returned, else a copy of the new item is made by the "copy_item" function and it is added to the set, and 1 is returned.

replace: receives a pointer to a Bibliog_item, checks whether it exists (uses the compare function of the item), if it exists it is replaced by the new item that has identical key attributes but may have the rest of the attributes modified, and 1 is returned. If it does not exist 0 is returned. This function could be integrated with the "add" function, but we thought that it is safer for the user to separate the concepts of adding an item (key attributes that do not already exist), and replacing an item (key attributes already exist in the set, but other attribute values may be different).

remove: receives a pointer to a Bibliog_item, and checks whether it exists in the set (uses the compare function of the item). If it exists it is deleted from the set and 1 is returned. If it does not exist, 0 is returned.

specify: receives a pointer to a Bibliog_item, and if an item exists in the set that has the same key attributes, then a copy of this item is made (using the "copy_item"function) and a pointer to this copy is returned, else 0 is returned. The copy is made because the user program may want to modify an item that it has specified, and in this case it is safer to modify a copy and then substitute the original in the set by using the "replace" function.

retrieve_by_attrib: receives a pointer to a keyword, and by calling the compare_attrib function of each item, builds a linked list of all the items that contain the keyword in a specific subset of their attributes (uses the compare_attrib function of the items). Returns a pointer to the head of the list, or 0 if none of the items satisfies the condition.

retrieve_by_context: receives a pointer to a keyword and returns a pointer to the head of a list of items whose index (Assoc_array) contains the keyword, or 0 if none of the items satisfies the condition.

count: returns the number of the items in the set.

list: returns a pointer to the head of a list of all items in the set.

save_set: Overwrites the specified file and saves all the items (could use the save_item function of each of them). Maybe this function will be useful as public function.

read_set: receives a pointer to an input stream (associated with a file) and uses the "read" function of the appropriate item type to read all the items from the file into the set.

[34] Published

[35] Forthcoming

[36] Unpublished

[37] Non-paper

Are initialised by pointers to all Polyset instances that hold relevant items. For example, the "Published" set will contain pointers to all the polymorphic set instances that hold published items (books, periodical articles, collections, etc). The classes will use the member functions of the polymorphic set instances to provide aggregate operations like:

count: returns the number of the items.

list: returns a pointer to the head of a list of all items.

retrieve_by_attrib: receives a pointer to a keyword, and returns a pointer to the head of a list of items that contain the keyword in a specific subset of their attributes, or 0 if none of the items satisfy the condition.

retrieve_by_context: receives a pointer to a keyword and returns a pointer to the head of a list of items whose index (Assoc_array) contains the keyword, or 0 if none of the items satisfy the condition.

[38] Bibliography

Will contain pointers to instances of classes 34-37 and provide the same operations as those classes, but on all the bibliographical items that exist in the database.

A detailed description of the class interfaces is given in "APPENDIX C: User manual for the classes".

2.3. How the design meets the requirements

In section 1.2.2 of the previous chapter we outline the facilities that we would like to offer to the user.

Since the complete program can be very clearly separated to two different pieces, the classes and the client program that uses them, there must be a separation of the tasks and the responsibilities between those two pieces as well. This chapter refers to the design of the classes, and it is essential to find whether the facilities offered by the classes are adequate for the client program to implement the specified functionality.

1. *Operations on the structure of the database (navigation)*

This kind of operations has been left entirely to the client program, as it is a very high level decision and by incorporating them in the classes, we would make the classes less flexible and more tightly coupled to the specific user view of the system.

2. *Update operations (add, modify, delete)*

2.1 Bibliographical items can be added and deleted by using operations provided by the polymorphic set (Polyset) class: "add" will add an item, "remove" will delete it. To modify or remove an item, you have to specify it first. This could be done by calling "specify" of the polymorphic set class. By using "modify" of the Bibliog_item class, you can modify an item.

2.2 Access to the index of each item is easy since the associative array is public and its member functions are accessible from the user program. The function "put" can be used for adding new keywords and modifying existing ones. Deleting is achieved by "cancel".

3. *Retreive operations*

3.1 By using the "retrieve_by_attrib" function that is provided by all the set classes [33-38], you can retrieve the items whose certain attributes contain a given keyword. The keyword does not need to be the complete value of the attribute and it does not matter if it is in capital or small letters. It is matched against certain attributes that are considered to be important for retrieval purposes. Those certain attributes can be decided by the function "compare_attrib" of the Bibliog_item class, which search some of the attributes, depending on the specific item type, for the keyword.

Retrieval based on indexes can be achieved by using the "retrieve_by_context" function, provided by all the set classes [33-38]. This function can use the "at" function of the Assoc_array class, that retrieves data from the index. In this case, the keyword must be the complete word; the type of letters (small-capital) again is not important. A function that displays all the contents of an item, is the "print_index" of the Assoc_array class.

The client program can combine the two kinds of retrieval, to find items that contain a keyword either in an attribute field or in their index.

3.2 All the set classes [33-38] contain "count" functions that return the number of items in the set.

4. *Various operations.*

4.1 In order to move an item from unpublished to forthcoming (or from forthcoming to published), the client program has to use the "make_forthcoming" function of the specific item, that returns the corresponding forthcoming item. Then it has to delete the unpublished item from its set (by using the "remove" function of the Polyset) and add the new forthcoming item to its set (by using the "add" function of the Polyset).

4.2 Listing all items in a set is easy, as all set classes provide a "list" function that could be used in combination with the "print_attrib" of the Bibliog_item, that prints an item.

4.3 Seeing the index of an item is simple by using the "print_index" function of the Assoc_array.

4.4 The Bibliog_item class could also easily equipped with functions that store and retrieve the exact position in the library of a particular item instance.

A final part of the requirements which is not explicitly stated is the capability of storing and retrieving the data from disk. The way that this can be done is: when the client program declares an instance of a Polyset associated with a filename, it can load the set with the contents of the file, by calling the "read_set" function. All modifications remain in the main memory, until the destructor of the set is called. Then the file is overwritten by the (new) contents of the set. Alternatively, you can cause the disk to be written before the destruction of the set, by using the "save" function of the Polyset. However a Polyset class instance can be declared without a filename, and in this case the set is not connected with a file.

2.4. Justifying the design

The previous chapter identified the items that our library will accommodate. The model of the system is mainly based on the physical structure of these items: for example we have shown that a multivolume work consists of volumes, a periodical of articles etc. This implies that we should keep these relationships in the design, but, as described in section 2.1, maybe this would lead us to system entities that have disadvantages similar to the real world entities: the hierarchical structure "consists_of" makes it easy to find the parts of an entity - eg. the articles of a periodical - but difficult to search for an article if you do not know in which periodical it is. We expect computers to do efficiently things that are expensive in the real world, not things that can be done easily without them. In this case, it seemed better to us to choose the system entities, not according to their physical structure, but according to their conceptual importance. The most important thing about any item in a library is that it talks about a specific subject. From this point of view, an article is coherent, a newspaper is not. This makes the article more important, and the newspaper just an attribute of the article. In this way, we can emphasize the items that the user will be most interested in and we keep the design simple.

At this stage we have a clear idea of the system entities that represent the items of the library. We observe that there are a lot of similarities between them, both in their attributes and in the operations that we would like to perform on them. It would be very convenient, if we could treat them in the same way, despite the differences. In this case the type checking system of the compiler raises an obstacle: no matter how similar these item types are, if they are represented by different classes, they

seem completely different to the compiler. The way to overcome this problem is provided by inheritance: it will allow us to deal with different item types in a simple and uniform way, and virtual functions will make sure that each item will behave in a different way, according to its type.

What is the commonality that all item types could inherit? They all are bibliographical items. Which are the qualities of a bibliographical item? It has some attributes, like author or title, which however, may be different for different kinds of items; it has a content, it talks about some specific subjects. In design terms, it has an index similar to the indexes at the end of books, which allows us to find whether an item talks about a subject and where in the item you can look for it. This is what all item types have in common, plus operations on them like find an item, print the item etc.

Can we go further and create more such relationships? For example we observe that items belong to one of four categories: published, forthcoming, unpublished, and non-paper. What is more, published, forthcoming, and unpublished items are generally different types of either books or articles. What this suggest is that we could use multiple inheritance to make the various item types inherit one of the published, forthcoming etc. classes, plus one of the book, article, or any other class that represents a general type of items. This approach has the advantage of showing explicitly the relationships between the concepts that the classes represent, and of offering a natural grouping of the items which we could use to provide different operations for different groups.

Do we need to incorporate these relationships in the design? This grouping would be desirable , if there were some operations that we would like to perform on instances of published items only, or on instances of all kinds of articles only, or on some other kind of items only. We found two operations in the requirements that could lead us to include more is_a relationships in the design: the functions that move unpublished items to forthcoming and forthcoming to published.

However when we tried to isolate the common attributes of all published, forthcoming, etc. items, we realised that there are always exceptions that make it impossible to have a base class with attributes showing its essence as an entity: in some cases there are attributes common to most of the item types but never to all of them, in other cases there exist common attributes but do not express the concept that they are supposed to express. For example, it would seem that all published items have a publisher, a publication place and a publication date, and these attributes could be the attributes of a published_item class. A forthcoming_item class could have only a

publisher and a publication place, the published_item could inherit it and add the publication date. This organisation would be ideal for moving items from forthcoming to published.

However, the real world is not ideal: periodical articles are published items but do not need to keep a publisher and a publication place, they only have a publication date. Could we say that the publication date alone holds the concept of a published item? Or leave the periodical articles out of the published_item category just because it is convenient? And then what about non-paper or unpublished? The concept of unpublished is an item that cannot have a publisher's name a publication place and a publication date. How can we express this with inheritance? Maybe we should remember the warning that inheritance must not be overused. Conclusion: keep the design as simple as possible if this does not affect its functionality.

Now we can define a set to contain abstract bibliographical items. Different instances of this polymorphic set will contain instances of different item types. For example, one set instance will contain all monographs, another all collection articles, another all microfilms, etc. The polymorphic set is the heart of the system. If we design these sets to be independent of the layers above them, they will be able to be used by a client program as building blocks, providing operations tailored to different needs and different views of the same database.

The next step is to group these sets together, in order to create bigger sets that support operations on all published, forthcoming, unpublished, and non-paper items. In this case inclusion is the relationship we are looking for, not inheritance; it seems that a book, for example, is_a published item, but we are talking about sets now, and a set of published items has_a set of books. The set of published items is actually a client of the set of books and uses its facilities.

Alternatively, instead of making classes that represent the sets of published, forthcoming, etc. items, we could leave this responsibility to the user program. This approach emphasizes the idea that the polymorphic sets are the heart of the system and everything above them is closer to some particular need and view of the database. It also reduces the amount of work for the classes, but increases of course the effort needed to implement the user program. If we choose this approach, we must add a variable in the polymorphic set class to hold the type of each set instance, in order that the user program will be able to select the appropriate sets. In our case, this variable can have the values published, forthcoming, unpublished, and non-paper.

Which of the two ways will we adopt? It seemed to us that the division to published, forthcoming, unpublished and non-paper is a rather important one and deserves to be represented explicitly by individual classes.

At this stage we must ask ourselves how the items will be stored and retrieved from the disk[5]. We would like these operations to be transparent to the user program. Each item can have a function that will store its contents in a file and another that will read the file and restore the contents (for a more detailed description about these functions, see section 3.2 and APPENDIX C). The file itself can be defined to contain all the available bibliographical items, or we can have different files for published, forthcoming, unpublished, and non-paper items, or we can have different files for each item type: a file for books, a file for magazine articles, etc. We chose the last option, because it is simpler as all the items in the same file have the same structure, and it seems to be more flexible: with a small effort you are able to regroup the sets of the existing database according to new needs. For example you could easily implement a new set of articles, that will use the existing sets of magazine articles, newspaper articles, forthcoming articles, etc. to provide operations on all article-type sets of the library. By associating the files to the different item types, we make the sets of the item types independent of the layer that uses them (in this case the published, forthcoming, etc. sets).

A problem that arises is how to update specific items on disk whose contents have been changed. For example, it is easy to append the file when the user adds an item to the corresponding set, but how will the file be updated in case the user deletes or modifies an item? A simple solution is to overwrite the file with the new contents of the set only once, when the destructor of the set is called, or when the "save_set" function of the set is invoked.

A subtle point however is that, although we must be able to associate an instance of the polymorphic set with a filename, we must also be able to declare an instance of a polymorphic set that has not any connection with the disk. This will make the polymorphic set class more flexible, and it will be possible to use it for intermediate results.

So far, we have described the relationships of the item types and the sets that hold them, but what about the structure of the item types themselves? Each item type will inherit the functions declared in the abstract bibliographical item class, define them if needed, and declare its attributes, which will be instances of the Attribute class. The virtual functions will allow us to take full advantage of inheritance, as they adapt themselves to the object they operate on.

Despite the fact that virtual functions are very helpful and convenient, sometimes they try to be too helpful and the program that uses the classes cannot control them. For example, we include in the Bibliog_item class a virtual function "print", that will send to a specified stream the attribute names and the attribute values of any item. Since each item type redefines the function, the same "print" will always give the right result, no matter on what item type it is applied. However, what we gain in genericity, we loose in control of the details. The "print" function is not at all flexible, it will always have the same format. The user program cannot change this format, omit a specific attribute, or show only the attribute values without the attribute names. So we need a more flexible approach. A solution could be to include a function "list_attrib" that returns a pointer to a linked list of all attributes of an item. The user program can now intervene, and decide what to show to the end user, but it does not know the name of each attribute (eg. author's name, or title, or number of edition, etc.). This can be a problem when it deals with more than one item types at the same time. The question is: could we combine the advantages of the two functions? We could store the attribute name with each attribute instance, but, if we do this, we might as well keep a variable in each item showing its type, and that is what virtual functions are supposed to get rid of. So maybe there is not a single perfect solution. Then is the solution to include both types of functions, virtual and general, and conventional and flexible, where it seems necessary?

It seems that what we have touched is the top of an iceberg. More thinking reveals the true dimensions of the problem:

The functions that the bibliographical item class provides, determine the nature of the user programs. For example, if you read section 2.2 about the class interfaces, you can see that a lot of functions are connected to the standard input - output. This betrays a subconscious idea that we have about the user program: we know that we will not attempt to make a windows interface since the target of this project is not a complete and nice user program, but a very basic one.

The initial design of the classes, however, recognised this problem and the solution that was given is to declare all the attribute instances in the public part of the item classes. This would give maximum flexibility in the way that the user programs update or examine the attribute values. Since we do not know at this moment how the user programs that will be implemented in future will be, maximum flexibility seemed a good approach.

But, while we were implementing the classes, we realised that by declaring the attributes as public, we violated information hiding, and we gave to the user programs

too much freedom: by explicitly changing a key attribute of an item, the structure of the set that this item belongs to may collapse.

Whose responsibility is it to be careful? The classes or the user program? One thing is clear: it is difficult for the classes to be safe to use, and provide maximum flexibility at the same time. Maybe we have reached the limits of our capabilities as programmers, or maybe this is a reflection of a more general rule saying that you cannot have it all in life.

It seems that we have two options: the realistic one is to declare the attributes public and warn the user program to be careful; the romantic one is to protect the user program from abusing the operations by declaring the attributes as private, and let future user programs expand the classes according to their particular needs of functionality.

We will choose the second approach. It seemed to us that information hiding is like moral rules: there are always good and realistic reasons to break them, but sometimes you have to be romantic. So the decision was to sacrifice flexibility for our ideals.

Romantic decisions however are harder to carry out: The attributes are private now, but can be moved to public at any moment without affecting the rest of the code. The opposite is not true: if we had declare them public, we might have used them in places that would not be able to access them as private, so it would not have been as simple to move them from public to private.

Two special kinds of items are the microfilm and the items that can be translated. What they have in common is that they can have a varying number of attributes. Microfilms will include details about the book, the newspaper article, or the periodical article that they refer to; items that can be translated may or may not include details of the original. There are two ways we can solve this problem. The first is inheritance: for example microfilm could inherit a book or an article. However, we do not think that this issue deserves special care, or justifies further complexity of the design, so we will adopt the second solution[6]. That is to use the classes of book, periodical article and newspaper article to keep details about the microfilm. The microfilm class will have pointers to these three classes in the private part, two of which will not be needed and will always be 0. For the translated items, we will implement classes that will be able to hold details about the original items and keep in every item that can be translated a pointer to the corresponding class with details about the original.

2.5. Between design and implementation

The most important decisions concerning the implementation at this stage, have to do with the polymorphic set and the associative array (Polyset and Assoc_array classes). As efficiency is not our first priority, we will use simple data structures to implement these abstract data types: the polymorphic set will be implemented as a sorted array and the associative array as a hash table.

There are reasons for the selection of those particular data structures. First of all, a sorted array means that all the retrieval operations, including listing all the items of the set, will present the items in a specific order depending on their attributes, which is desirable. Second, when we retrieve items whose index contain a given keyword, since we have one index for every item, if there are n items in the set we must enquire all n indexes sequencially. This of course can be improved, but for now what we need is a fast index that will not slow down this sequencial probing even more: a hash table will enable to search the indexes of n items in O(n) time. A final reason is that we have not implemented these data structures before. A polymorphic set that uses a binary tree has been taught as an example so we tried a different way; what was interesting for us about the hash table is the collision strategy: open addressing - linear probing[7].

A sorted array of pointers to bibliographical item instances, can be used for returning to the user all the items in a specific order, as well as finding whether an item exists in the set. The items will be sorted according to their key attributes: those that include authors and titles in their attributes, will be sorted according to author names, and for the same names according to the title. The same holds for the rest of the key attributes (eg. publication date). Binary search will be used to find an item in the set whose key attributes are known. This means O(lgn) time, when the set contains n items.

The set instances may be associated to a filename. In this case the filename will be given to the constructor as a parameter. However, it will be useful not to restrict the operation of the set by making the association with files compulsory, so if no filename is given, the default will be to create a set instance without connection to a file. This set might be used to store and operate on intermediate results, for example in the case of retrieval based on more than one keywords.

Another issue is the size of the array. We cannot know this size, as it depends on the number of items on the disk and the number of items that the user might add to the set. A solution could be to start with a default size and increase it when it becomes

insufficient. This of course, will be transparent to the user of the class.

The associative array will be implemented as a hash table consisted of an array of pointers to records. Each record will include two pointers to strings, one for the keyword, one for the page number(s). We will use open addressing, and linear probing as a collision strategy[7]. The size of the hash table will be fixed.

We do not claim that this is the best possible data structures for the implementation, but they are not expensive to change as long as the interfaces of the classes are kept the same.

3. IMPLEMENTATION

This chapter is about the implementation and testing of the classes, that constitute the basic blocks of the system and the main part of the project. The design and implementation of the user program that demonstrates how the classes work is discussed in the following chapter.

3.1. What has been implemented

As we said in section 1.1.5, we consider the final user program to be important, and we would rather implement this than some of the classes, if time is not enough. Time has not been enough and we had to choose, but even if it had been enough we would still prefer to do something more interesting than implementing the rest of the classes.

The reason is that the classes that we have not implemented are classes for some of the bibliographical item types, whose code is similar - almost identical - to the item types that have been implemented.

Section 1.2.1 recognises 22 item types, of which we have implemented 9: 3 published items (book, monograph, periodical article), 2 forthcoming (forth. book, forth. article), 2 unpublished (research, theses), and 2 non-paper (microfilm, software). We have chosen those 9 items so that they include all items that have some particularity or difficulty: the book type is an example of items that can be translated, the microfilm is a special case because it can hold a book or a periodical article or a newspaper article, the forthcoming book is an example of forthcoming item that can be moved to published, and the departmental research is an example of an unpublished item that can be moved to forthcoming.

Everyone of the 13 item types that have not been implemented is identical to one of the 9 items that have been implemented. The only difference will be the number and the names of their attributes.

The fact that we selected item types from every category (published, forthcoming, unpublished, non-paper), makes it possible to build a user program that has the

same structure and offers the same functionality as if all the item types existed, and that can easily incorporate the rest of the items when they become available.

3.2. Obstacles encountered and solutions given

Many of the decisions about the program have been taken during the implementation stage. This does not necessarily mean that the design was inadequate. Brooks[8] suggests that some of the decision - making must be left to the implementors. Even in the case of a very detailed design, because of the nature of the software development process, it is possible to find during the implementation that you have to change some parts of the design. This is a summary of the most important implementation issues of this project.

Let's be tidy and begin with storage allocation. Whenever new storage is allocated, the operation is checked and in case of failure, the program is stopped as a safety measure against corruption of other memory locations. The only exception is when we allocate storage like this:

```
char *new_value = strcpy (new char [strlen(value)+1], value);
```

as it seemed to us that it is the responsibility of the "strcpy" function to check for zero pointers before copying the data, and to do something sensible if any of the pointers is zero.

An important issue was to read a string from a file where it has been saved. The string is written to the file as a sequence of characters, can have any length, and is possibly followed by other strings. The problem is how to read the correct number of letters for each string, without leaving any characters in the file, or taking any characters from the next string. A solution could be to have a special character that denotes the end of a string. This of course might restrict what characters we can have inside a string, except if we make complicated read - write routines that guarantee transparency. The solution we adopted is to write in the file the length of a string before the string itself, so that we know exactly how many characters to read. The next character written in the file after the length of the string is a space, so that we can read the length like this:

```
int length;
(*file) >> length;
```

And then:

```
char dummy;
char *str = new char [length + 1];
file -> get (dummy);     //take the space char
file -> get (str, length+1);     //take string
```

We used similar techniques not only for strings, but for the associative array class and for the polymorphic set class as well. Before writing the items of the associative array, or the members of the set, we write their number, so this is the first thing that the read function gets from the file. Special cases are the microfilm, where we have to save the kind of item that the microfilm is about, and the items that can have either authors or compilers or editors, where we have to write to the file the type of the creator of the item.

Another problem concerning the function that reads the contents of a polymorphic set from a file is the following: the set is defined to contain instances of the bibliographical item class, which is an abstract class as it has declarations of virtual functions that are not defined in the class. One of these functions is the "read_item" that reads an item from the disk. The "read_set" function uses that function in a loop like this:

```
read number_of_set_elements from file;
for (int i=0; i < number_of_set_elements; i++) {
     Bibliog_item *item = new Bibliog_item;
     item -> read_item (file);
     add item to set;
}
```

This does not work, because the Bibliog_item class is abstract and you cannot declare an instance of it. On the other hand, we want the set to be polymorphic, so we cannot use the classes that inherit the Bibliog_item (Book, Microfilm, etc) in its position.

The solution we gave, is as follows: the set does not know what kind of bibliographical items (books, microfilms, etc) exist in the file it is about to read, but the program that uses the set does. So the user program can pass a Bibliog_item pointer that points to the right type of item to the "read_set" function, which can now use the

"read_item" function of the item. Not the most elegant solution, we must admit, but one that works. A consequence of this solution is, that if we make the "read" operation transparent to the user program by calling the "read_set" from the constructor of the set, we have to add another parameter to the constructor: a Bibliog_item pointer that will point to a newly created item of the type that the set will accommodate. This seems to be quite nasty, so we will leave the responsibility of reading the file to the user program. The only thing we could do about transparency was to call the "save_set" function from the destructor of the set.

The polymorphic set has been implemented as a sorted array, so we must choose a way to overcome the size limit of the array. A sensible solution is to add a constant to the array size each time that it becomes inadequate, create an array of the new size, copy all the elements of the old array to the new one, and delete the old array.

Another controversial issue concerning the polymorphic set is about the key attributes of the set elements. Should we allow them to be null or should we demand that all of them must have non-null values for an item to be inserted into the set? If this was a relational database this question would not exist, because we would not have a choice. We thought that since the key attributes are just a convention we made, and are not declared as such by the nature, they have the right, as any other attribute, to be ignored or forgotten by the user. We can afford this approach as we consider the null value to be like any other value: you cannot enter in the set more than one items with all their key attributes null, and when you print the contents of the set, the null key attributes are printed as well (there is a special character for them), so an item with null key attributes will be noticed and maybe removed. However, the opposite approach may be more valid from a logical and mathematical point of view.

Some bibliographical items may be moved from unpublished to forthcoming and from forthcoming to published. The way this is done is explained in the design, but there are more details to be determined. Let's use an example: we want to write a function that will make a published book from a forthcoming book. Clearly, this function belongs to the forthcoming book class. It will create a published book instance, will copy the values of the attributes that the two items have in common from the forthcoming to the published item, and prompt for the rest of the published book attributes. The problem is that the attributes of the published book are declared private and cannot be accessed by the forthcoming book class. One solution is that the function will call another function of the published book class with all the values of the common attributes passed as parameters, and this last function will do all the work. This is obviously messy, but we can do better, if we adopt a different approach:

declare the function of the forthcoming book class friend[6] to the published book class, so that it has access to its private variables. Unfortunately, this solution has the disadvantage of having to compile the published book class every time you compile the forthcoming book class, even if you never intend to use the function that make the forthcoming book, published book.

An interesting point is about a problem that we expected to have, which turned out not to be a problem at all. Some of the functions of the various bibliographical item classes (book, collection, theses, etc), create a new instance of the item, and set its attributes. For example the function "copy_item" creates a new instance of the particular item type, and copies all the attributes of the current item to the new one in order to make an identical copy. The attribute "title" of the new item is set like this:

```
new_item -> title -> set (title -> check ());
```

The first "title" refers to the new item instance, the second to the current instance. The problem could be: does this class, that represents the current item, has access to the private variables of a different item instance? If it does not, we are in trouble. Fortunately it does; the class does not represent one particular item instance, but the abstract concept behind all instances[9].

The last problem that we present here, has to do with the sets of published, forthcoming, unpublished, and non-paper items. Should we declare the various item type sets inside their corresponding set of published, etc. items? We think that this is not a good idea, since it results to reduced flexibility, so we decided to pass pointers to the sets of item types, to the constructor of the corresponding set of published, etc. items.

During the implementation we tried to anticipate possible extensions of the classes in future, and to make things as easy as possible for the people that will change them. It would seem that this is mainly a design responsibility, but it became clear that careful implementation can be a great help. As an example, consider a function of the published set that lists all the items of the sets that belong to the published set (books, collections, newspaper articles, etc). This function obtains one linked list from each set, that contains all the items of the set. It has to link the lists together so that there is only one list containing all the published items. One could think something like:

```
tail (books_list) -> next_item = collections_list;
tail (collections_list) -> next_item = period_articles_list;
.......
```

and so on. This would not work, because if a set has not any elements, it will not return a pointer to an item, but it will return 0.

A solution is to make nested "if" statements that will examine all possible combinations of the lists, to avoid trying to link a zero. In our case it would be easy to do, since we implemented only 3 item types that belong to the published set. But it would be very tiring and possibly difficult to expand this function, if you add many more sets of different item types to the published set. The solution we gave is simple, more elegant, and easy to expand. All that the "list" function has to do is to put the pointers to the lists in an array, and call the following function:

```
Bibliog_item *link_lists (Bibliog_item **heads)
{
     // link the lists whose heads are pointed by elements of the
     //'heads' array. Be careful: the pointers may be 0.
     int tl = 0;
     int hd = 1;
     Bibliog_item *first_head;

     //member_num is the size of the 'heads' array
     while (hd < member_num && heads [tl] == 0) {
          tl += 1;
          hd += 1;
     }
     first_head = heads [tl];  //first list with at least one element

     while (hd < member_num) {
          while (hd < member_num && heads [hd] == 0)
               hd += 1;
          //we now have the next list that is != 0
          if (hd < member_num) {  //link the two lists
               //'tail' returns the last item of a list
               (tail (heads [tl])) -> next_item = heads [hd];
               //advance counters
               tl = hd;
               hd += 1;
          }
     }

     return first_head;  //head of the very first list that was != 0
}
```

3.3. Testing strategy

The most important factor that influenced the testing and debugging is that we used classes to implement the system. It is amazing how easy it becomes to test a program if you do it piece by piece, starting from the most basic pieces that do not depend on or use other modules. Once you have tested a class, you can rely on it and minimise the possible causes of an error.

Sometimes you are not sure whether the error is caused by the module that you test, or by the test program itself. In this case, it may be possible to substitute the class that you are testing by another that you have already tested, or with a very simple version of the original class that is certain (!) not to contain errors, and test the test program first.

The way we approached testing and debugging is simple: the test programs created a simple interface that enables you to invoke the member functions of the class being tested in any order, give any arguments that are needed at run time, and check the results on the screen.

The polymorphic set has been tested by two different test programs. The first one used as set elements instances of a very simple version of the bibliographical item class, that was just a string with operations to compare it and copy it, to make sure that the set works as expected, and the second used a normal bibliographical item class (the monograph class) to check the set under "real" conditions.

Of course the bibliographical item classes had been tested before. We made separate test programs specially for the functions that move items from unpublished to forthcoming and from forthcoming to published.

The only classes that have not been tested in the way described above are the Published, Forthcoming, Unpublished, Non_paper, and Bibliography classes. The reason is that the only important function in all those classes is the "link_lists" function, which is used by most member functions of the classes. So we thought that a program testing this particular function was enough.

3.4. Interdependencies of the classes

Figure 3.1 shows the classes that have been implemented as part of the basic system (not the user program classes). An arrow from one class to another means that the first class includes the second.

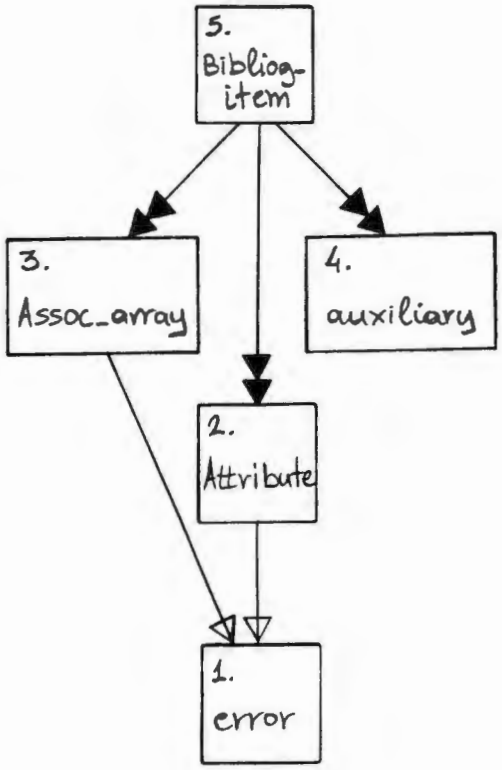
A class may include another class because it uses facilities that the second class offers, and the first class cannot work or be compiled without the second. For example, the Bibliography class includes the Published class because it uses the member functions of the Published class. This kind of relationship is represented by a single white arrow.

A class may also include another because the second class conceptually belongs there. For example the Bibliog_item class includes the Assoc_array class, the Attribute class, and the auxiliary class, although it does not use any of them. However all the items that inherit the Bibliog_item class use all three classes, so it seems convenient and semantically right for these classes to be included in the Bibliog_item class. Another example is the class Forthcoming that includes all the forthcoming item classes, although it does not use them. The fact that it includes them can make a user program simpler and clearer. This relationship is represented by a double dark arrow.

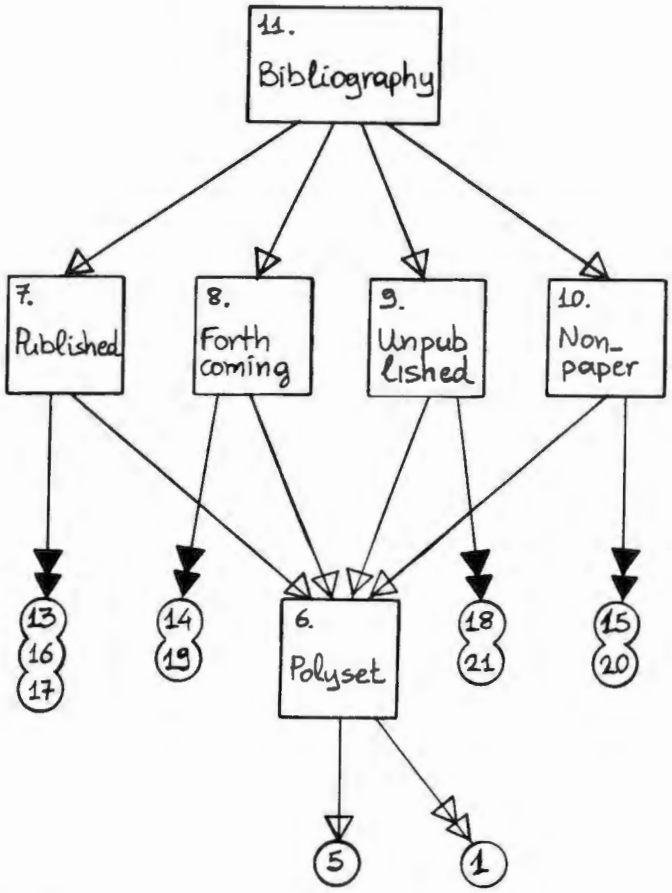
Another case is when a class uses a second class but does not need to include it, because it is implicitly included already. This is the case of the class Polyset that uses the class error, which is implicitly included because the class Bibliog_item includes "error" and is included in "Polyset". However all this is not very clear, so it seemed to us conceptually better to explicitly include the error class in the "Polyset" again. This relationship, where a class is used and is implicitly included but is also included explicitly a second time for reasons of clarity, is represented by a double white arrow.

Complex inclusion relationships may lead to redefinitions of classes. To avoid this, we find the classes that are likely to be redefined or that are included by more than one classes, and we protect them against redefinitions. The classes protected in this way are shown like squares, the rest of the classes like rectangulars.

a. Basic structure



b. Sets structure



c. Items structure

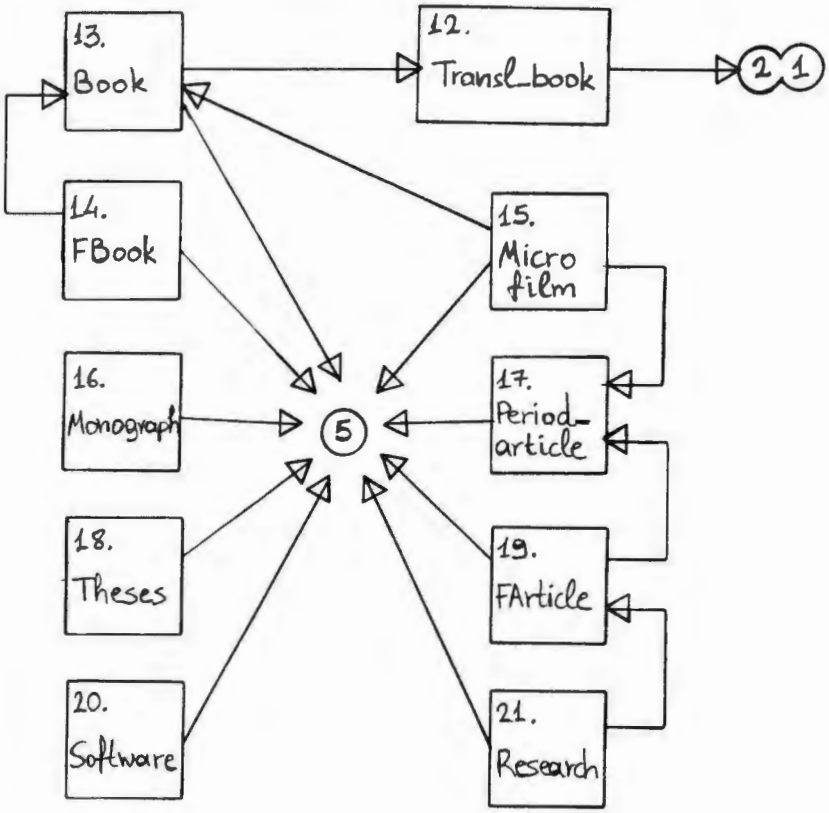


Fig. 3.1 : Interdependencies of the classes

4. A USER PROGRAM EXAMPLE

This is the most enjoyable part of the project. It is a pity we did not have more time for the user program, but considering that this program has never been the main target of the project, we are not unhappy with the result.

4.1. A short description

The final user program that has been implemented conforms to the requirements that have been outlined in section 1.2.2. It uses all the facilities that the classes offer, except that it does not move items from unpublished to forthcoming and from forthcoming to published. It is complete in the sense that it offers all the necessary update and retrieval operations, and it demonstrates one way that the classes can be used.

4.1.1. User view of the system

The user sees an hierarchy of sets, where the sets on the higher levels include some of the sets beneath them (see also figure 1.2). So going up means generalisation and aggregate retrieval operations on more sets, going down means specialisation. The lowest of these levels is a collection of sets of specific types of bibliographical items. At this level you can insert or delete items and specify an item. When you specify an item you change level and you can modify or print the specific item that you have chosen. You can go to an even lower level, to the index of the specified item that provides its own operations for update and retrieval.

Figure 4.1 gives pictorially the user view of the system. There are many positions where the user can be, but only 5 levels. Each level can have more than one positions, for example the level GROUP consists of the positions 1(published), 2(forthcoming), 3(unpublished), and 4(non-paper). The commands that are available depend on the level, and not on the position; all positions on the same level provide the same commands.

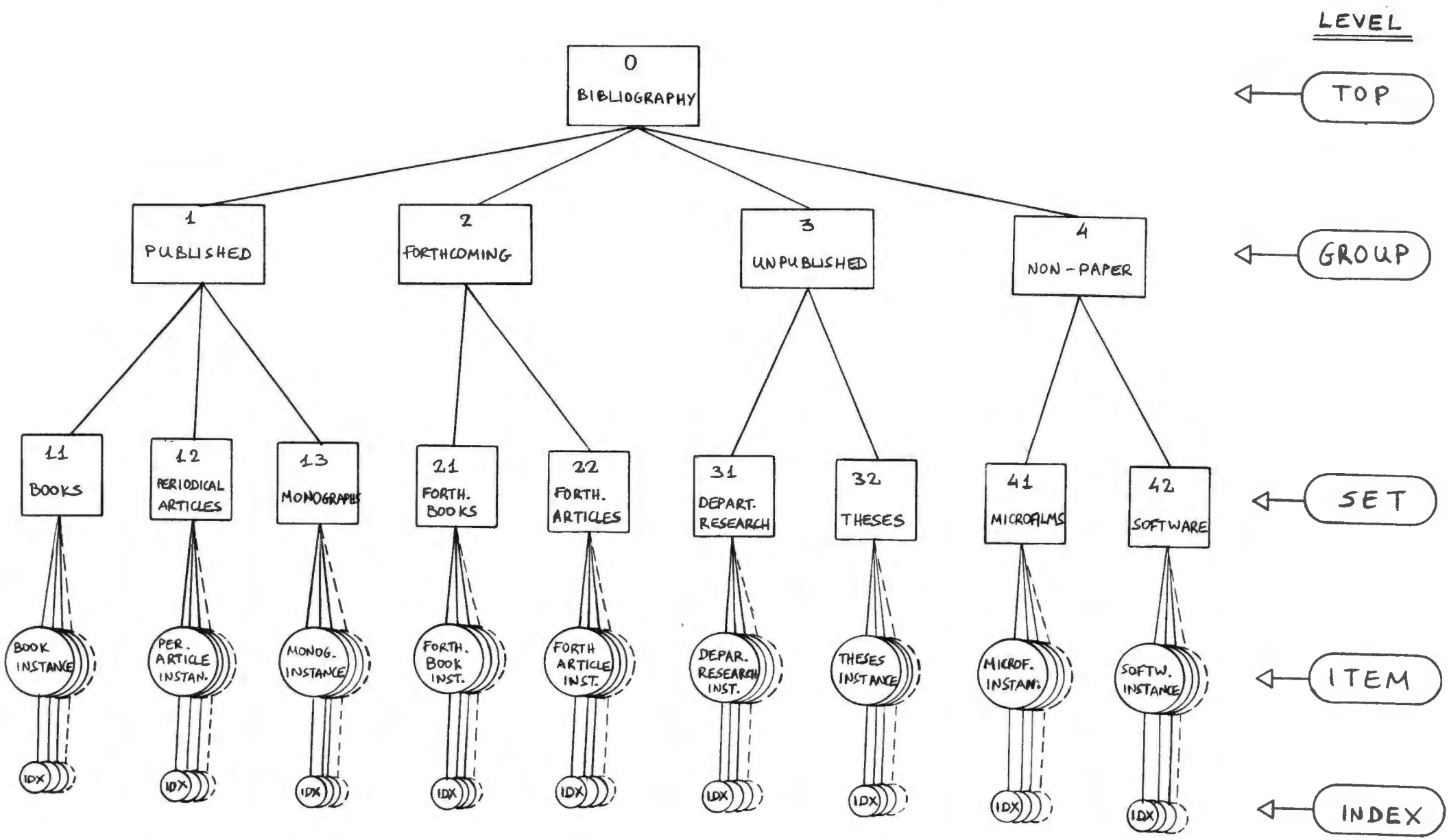


Fig. 4.1 : User view of the implemented system

The navigation is not like the way we navigate through directories, it is more direct and flexible: the user can move from any position to any other if he/she is on the level TOP, GROUP, or SET. However if you are on the ITEM or INDEX level, you can only go up to the corresponding position of the SET level, and then continue the navigation.

You can leave the program from any position of the upper three levels (TOP, GROUP, and SET).

4.1.2. Available commands

The program reads the disk automatically when it starts, and saves the contents of the sets when the user leaves the program. The operations that one can perform depend on the level (which can be seen on the screen as a prompt sign):

All levels :

? see all the commands available at the level you are, with a short description.

TOP, GROUP, and SET levels:

w see where you are.

b see the "children" below.

g <integer> change position to the position indicated by <integer>.

l list all items of the current position.

n see the number of items of the current position.

a <string> see all items of the current position whose one of the attributes contain <string>.

c <string> see all items of the current position whose index has the full <string>.

q leave the program.

SET level only:

i insert an item in the set of the current position.

d delete an item from the set of the current position.

s specify an item and change level to ITEM.

ITEM level:

p print the item on the screen.

m modify the item; <return> will keep the same attribute value, '-' will make the attribute value null.

r replace the original item in the set with the current version of the item (maybe modified).

i change level to the INDEX of the item.

u go up to the SET level.

INDEX level:

i <key> <val> insert a pair of strings in the index; if <val> is '-', <val> will be null.

c <key> see the "val" that corresponds to <key>.

d <key> delete the pair "key" - "val" that corresponds to <key>, from the index.

l list all pairs in the index.

n see the number of all the pairs in the index.

u go up to the ITEM level.

4.2. Design and implementation

It is obvious that the main responsibility of the user program is to provide the necessary functions to support navigation; the rest of the work will be done by the member functions of the classes. Another responsibility of the user program is to create an interface between the user and the member functions of the classes, that will permit the user to enter a command and activate a particular member function. Also it must receive the results and print them on the screen, when this is not done by the member function that has been invoked.

It seems that the structure of the user program will be like this: a main loop will implement the navigation, by receiving a command given by the user that selects the desirable position. Then a function that corresponds to that position will be called, which contains another loop with the available commands at this position. However, this structure means that we will have to repeat the same code over and over again, because most of the positions provide identical commands.

Since our aim is the quick development of a user program, why not try to use the advantages that object orientation provides towards this direction? Figure 4.2 shows the structure of the user program. The square boxes represent classes that we have already implemented as part of the basic system. The rectangular boxes represent the classes that are part of the user program. Each of the user program classes is associated with one of the basic system classes, and provides an interface between this class and the user. Some navigation facilities are provided by the class Position, which is inherited by all the user program classes.

The main advantage of this structure, is that we need only one loop for all the possible positions now. Since a pointer to the class Position can also point to any other class that inherits "Position", we can have a loop that activates the commands of the current position, and one of these commands can assign a class representing a different position to the pointer used in the loop:

```
char ch;
Position *current = bib;  //bib = default position at TOP level
do {
     cout << current->level << " > ";   //prompt for command
     cin >> ch;
     if (ch == 'w') current->show_position ();
     else if (ch == 'b') current->show_children ();
     else if (ch == 'l') current->list ();
     else if (ch == 'n') current->number ();
     else if (ch == 'a') current->retrieve_attrib ();
     else if (ch == 'c') current->retrieve_context ();
     else if (ch == 'i') current->insert_item ();
     else if (ch == 'd') current->delete_item ();
     else if (ch == 's') current->specify_item ();
     else if (ch == '?') current->help ();
     else if (ch == 'g') {   //change position
          int pos;
          cin >> pos;
          if (pos == 0) current = bib;
          else if (pos == 1) current = published;
          else if (pos == 2) current = forthcom;
          else if (pos == 3) current = unpubl;
          else if (pos == 4) current = non_pap;
          else if (pos == 11) current = books;
          else if (pos == 12) current = per_arts;
          else if (pos == 13) current = monogrs;
          else if (pos == 21) current = fbooks;
          else if (pos == 22) current = farts;
          else if (pos == 31) current = resrch;
          else if (pos == 32) current = theses;
          else if (pos == 41) current = microf;
          else if (pos == 42) current = softw;
          else msgs.error_message ();
     }
     else if (ch != 'q') msgs.error_message ();
```

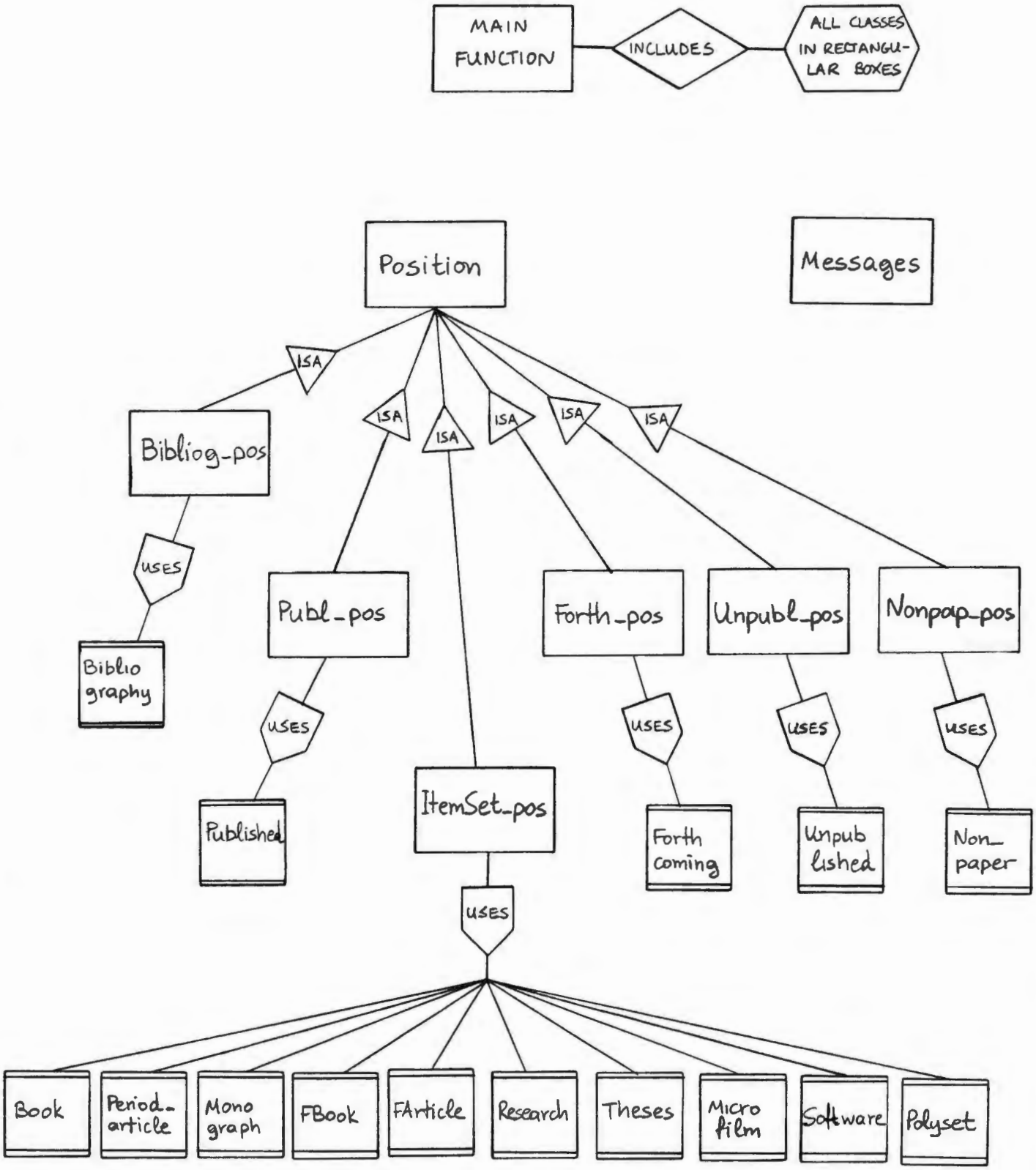


Fig. 4.2 : Structure of user program

```
}
while (ch != 'q');
```

This structure means that the class Position must declare all the functions that appear in the main loop. Some of them, like "show_position", conceptually belong to the class Position and are inherited by the rest of the classes without being redefined by them. Some have to be abstract virtual functions, because each of the classes that inherit "Position" has its own definition of them. These functions correspond to commands that are available to all positions, like "list".

However, there is a problem: what about the functions that exist at some positions but not at others? An example is the function "insert_item" which exists at all positions of the SET level, but does not exist at the positions of the GROUP and TOP levels. The solution is to define these functions in the class Position as virtual functions that print an error message. In this way, if a class that inherits "Position" does not redefine these functions, each time that you invoke one of those functions of this class, you will get an error message.

This is a bit messy, as the class Position has member functions that do not conceptually belong to this class, but it simplifies very much the user program, and it makes the main loop clear and comprehensive.

In figure 4.2 there is a rectangular box named "Messages". This is a class used by the main loop to provide error messages, and it implements our idea about how error messages should be. HCI experts suggest that error messages should not make the user feel as if he/she made a mistake[11]. On the contrary, the machine must assume the responsibility for not recognising what the user wanted!! This leads to overpolite messages like "I am extremely sorry I do not provide the command you have entered". It is our impression that this is not a user friendly message, as you would never expect this kind of behaviour from a friend. Overpoliteness creates a formal atmosphere that makes people feel frustration. What we need is casual behaviour that make the user feel comfortable and - why not - assume responsibility of his/hers mistakes. That is why our error messages depend on how many errors the user has made: the more errors, the less tolerant the message will be. The messages vary from "Sorry, my fault, I do not provide this command", to "You're tired, go home and try again tomorrow".

4.3. Other ways of using the classes

As we mentioned before, the aim of the project is not to make a carefully designed user program that covers every possible need of the user. If we had more time however, we would try to use the existing classes in order to implement a user program that would be able to:

- move items from unpublished to forthcoming and forthcoming to published. This would be done by invoking the corresponding member function of the item to be moved, but remember that all that this function does is to create a new forthcoming or published item. The user program would have to delete the old item from its set, and to insert the new item into the right set:

```
Bibliog_item *published_item = chosen_forthcom_item->make_published ();
corresponding_published_set->add (published_item);
delete published_item;          //'add' has made a copy
corresponding_forthcom_set->remove (chosen_forthcom_item);
delete chosen_forthcom_item;    //this is a copy of the item in the set
```

- retrieve items that satisfy conditions based on multiple keys. One way that this could be done is to repeatedly retrieve the items that contain one key, put them in a new set that is not associated with a file, and try the next key, until all the keys have been applied:

```
create an array of the keys;
create an array 'temp_set' of polymorphic sets;
initialise temp_set[0] by inserting all items of set you want to search;
loop (i)
     Bibliog_item *head = temp_set [i] -> retrieve_by_attrib (key [i]);
     insert all items of the list pointed by 'head' into temp_set [i+1];
end loop;
Bibliog_item *result_list = temp_set [i] -> list ();
```

A much better way of doing this could be to create another class that would receive the keys and the sets to be searched, and will offer member functions that will return the result of searching by attribute, by context, or a combination of both.

- Another point we consider important, is the possibility of creating different user views. At the moment, the user sees the sets of the bibliographical items as parts of the published, forthcoming, unpublished, and non-paper sets. Maybe it would be

useful to create a set using all the polymorphic set instances that hold articles in general, all types of book-like items, or whatever seems convenient. This would permit selective retrieval operations on some of the polymorphic set instances. Even more interesting would be the possibility of the user being able to create his/hers own views of the database at any moment he/she desires, save the arrangement on disk, etc. This would add considerable flexibility and would make the classes much more useful.

5. CONCLUDING REMARKS

We have made a program that works and that could be used to satisfy some very basic needs. This is where the project ends, but it would be an illusion to consider this as the end of the evolution of this program as well. This chapter tries to look back and see what we would have done differently if we could do it all over again, and what we would add if we had more time.

5.1. Improvements

Brooks[8] says that the question is not whether to make a prototype or not, but whether to plan for one, or to agree to deliver the prototype to the customer. The meaning of this is that even the most well-planned design can be improved, but these improvements can be seen only at a later stage of the software life - cycle. So, we include here some suggestions for possible future improvements.

a) *Speed:*

- When we use the "retrieve_by_context" function of the set classes, all the items are searched sequencially. For n items, since the time to check the index of each item is constant (hash table), we will need O(n) time to find the items that contain in their index one keyword. Maybe this could be improved by building an index that contains the entries of all individual indexes with pointers to the corresponding items.

- When we use "retrieve_by_attrib" we search for items whose any attribute of a specific subset of attributes matches the keyword. This function will call the "compare_attrib" function of the Bibliog_item class. Since this latter function compares the keyword to more than one attributes (otherwise it would not be useful), then the result will be sequencial search, since we keep the items ordered according to a single attribute.

A solution could be to build inverted tables and extend the functions provided by the classes to support retrieval based on many attributes by using the inverted tables. This would also permit us to efficiently retrieve items by giving keywords that correspond to various attributes. At the moment efficient retrieval can be done only if

you search by the first key attribute, since this is the one that mainly determines the position of the item in the array.

Another solution could be to make each item insert automatically into its index (associative array) the main words in some of its attribute values, and then check the index with the usual operations. This could work with attributes such as "title", but semantically there is no point in adding the publisher's name in an index that is supposed to keep the contents of an item.

b) *Space:*

- The items that can be translated (Book, etc.) have a pointer to a class holding the details of the original item. Space for this class is allocated even if the item is not the product of a translation. Maybe this space could be allocated after a user's response to a relevant question.

- The Microfilm class has three pointers to the three different classes that a microfilm can be about (book, newspaper article, periodical article). Two of them are always 0, so there is no space wasted here, but the fact that we keep three pointers while we need only one can be improved, maybe by using variant records.

- When we save the attributes of the items in a file, we also write a space after each attribute. This is not necessary, because the "read" function knows how long each string is going to be. The only space character that is important is, as indicated by the comments in the code, the space after the number that holds the length of the next string (see also section 3.2 and APPENDIX C). In addition, the fact that we write the length of a string plus a space character before writing the string itself, consumes some space. Maybe we could denote the end of the string with a special character, and modify the "read" and "write" routines to make this transparent to the user (to permit this character to appear in a string as well).

- The associative array (index) of the bibliographical items has a fixed size. This is wasteful of memory (pointers that may not be used) and constrains the number of entries the user can have for a particular item, but it is easy to change by using a different data structure.

c) *Miscellaneous:*

- When you save a polymorphic set instance to a file, if the file exists it is overwritten by the current contents of the set. We used a flag to avoid overwriting the file if the contents of the set have not been changed, but the problem remains: all the file will be overwritten, even if only one item in the set has changed.

- When you retrieve items from a set, a linked list is returned containing the items. This is not a good way, since it gives away the private data of the set. If the user program changes one of the items in the list, the structure of the set may collapse. One solution could be the list to contain copies of the items in the set, but this is obviously extremely wasteful of memory.

5.2. Future extensions

We tried to make the classes easy to change and to extend. In future, more functions could be added to support:

- windows interface
- retrieval based on multiple keys
- information about the position of a particular item instance in the library.
- facilities to create references in text.

Also more item types could be added to the existing ones. We mention 13 more item types with their attributes in APPENDIX B, that have not been implemented. Their structure however is identical to the 9 item types that have been implemented. In future, there will possibly be a need to define more item types that are now beginning to appear: CD ROMs is one example, and electronic publishing another.

5.3. Assessment

The assessment of something you have done is most important, because it reflects your experience, your mistakes, the things you have learnt, and those that will remain in your memory and guide you in future decisions.

5.3.1. Assessment of the methods used

One of the main factors for choosing this project was the desire to become familiar with object oriented techniques. It would be arrogant to claim that I now know

what object orientation is, and even more arrogant to attempt a critique of the techniques and the facilities I used in this project. However, I would like to share some thoughts that sprang from this effort, and to express a few questions that are, very probably, result of my ignorance and lack of experience.

It has been very natural to use the concept of objects in this project; I think it would be much more difficult to design this program without it. Encapsulation and information hiding made testing and debugging, which is usually painful, easy and simple (as described in section 3.3). Inheritance, polymorphism, and dynamic binding saved a lot of time of the implementation of both the classes and the user program, by removing the need of writing similar pieces of code many times.

I am not going to repeat all the advantages of O.O. programming that one can find in every relevant book. I would rather concentrate on the features that I found most helpful, and on the questions that some features generate.

Cox[9] says that O.O. means different things to different people: to software engineers O.O. means encapsulation, abstraction, and information hiding, to researchers of the academic community it means inheritance, polymorphism, and dynamic binding.

What does O.O. mean for me? Not surprisingly, I find the concepts of information hiding and encapsulation extremely useful and convenient. From this point of view, O.O. imitates the power of abstraction that the function has, but at a higher level. This means that some decisions have been already taken by the people that decided how these "objects" will work, so when compared to the concept of the function, the object seems to have sacrificed some flexibility for more power. The result of this combination seems to be twofold: a) the difficulty at the initial stages of the O.O. design: choosing the objects and defining their interface, and b) once you have overcome the problem of finding the right objects and the right interface, you enjoy the power of O.O.: you have cracked the problem to manageable pieces that are relatively easy to implement and test.

Inheritance, virtual functions, polymorphism, and dynamic binding - as implemented in C++ - could be seen as an effort to achieve abstraction at an even higher level. However I think that their success is limited compared to the function, maybe because the higher the level, the more complex the concepts become, so the more difficult it is to factorise concepts and achieve abstraction.

A more concrete example: when I was implementing the classes of the various types of bibliographical items, I found myself repeating the same code for all the item

types. The main difference was the number and the names of the attributes that each item type has. Ideally, the code of the member functions should be in the Bibliog_item class, which all item types inherit, and then each item type would just declare its attributes and define any particular function that this type has. Maybe a different design could permit this structure, but I think that the fact that this design is not obvious reinforces my point.

I must be fair and say that none of the above is unexpected: the more powerful a tool is, the more specific the application, the more dangerous it can be to use it, and the more specialised the knowledge you need to control it; power does not distinguish between good and bad use, it is the knowledge of using it that does. The fact that I felt that something is missing for inheritance to be more efficient, is most probably due to my way of using it. However it is this feeling that makes inheritance more interesting to me than information hiding and encapsulation, and it could justify the interest of the academic community as well.

5.3.2. Assessment of the project

The most important thing that this project gave me, is confidence to my decisions. I must admit it is the longest program I have written, and it has been very encouraging for me to see it work. It has also been a good experience on object orientation, databases, and the various stages of the software development.

Now I have more questions than before, but that is how learning works; next time, maybe I will find some of the answers.

BIBLIOGRAPHY

1.

On-line Unix manual, "addbib" - "indxbib" - "lookbib" - "refer" by Mike Lesk - "sortbib" by Greg Shenaut and Bill Tuthill - "roffbib".

2.

J. C. Alexander, "Tib, a TEX bibliographic preprocessor".

3.

Kate L. Turabian, A Manual for Writers of Research Papers, Theses and Dissertations, first British edition, London, Heinemann Ltd, 1982.

4.

Darrel Ince, Object Oriented Software Engineering with C++, Berkshire SL62QL, McGraw-Hill, 1991.

5.

Paul M. Chirlian, Programming in C++, Ohio 43216, Merrill Publishing Company, 1990.

6.

Bjarne Stroustrup, The C++ Programming Language, 2nd edition, Addison-Wesley, 1991.

7.

R. Kruse, B. Leung and C. Tondo, Data Structures and Program Design in C, Englewood Cliffs N.S.07632 USA, Prentice Hall, 1991.

8.

Frederick P.Brooks, The Mythical Man-month, Addison-Wesley, 1975.

9.

Brand J. Cox and Andrew J. Novobilski, Object-Oriented Programming, 2nd edition, Addison-Wesley, 1991.

10.

Bertrand Meyer, Object-Oriented Software Construction, Hertfordshire HP24RG, Prentice Hall, 1988.

11.

Ian Sommerville, Software Engineering, 3rd edition, Addison - Wesley, 1982.

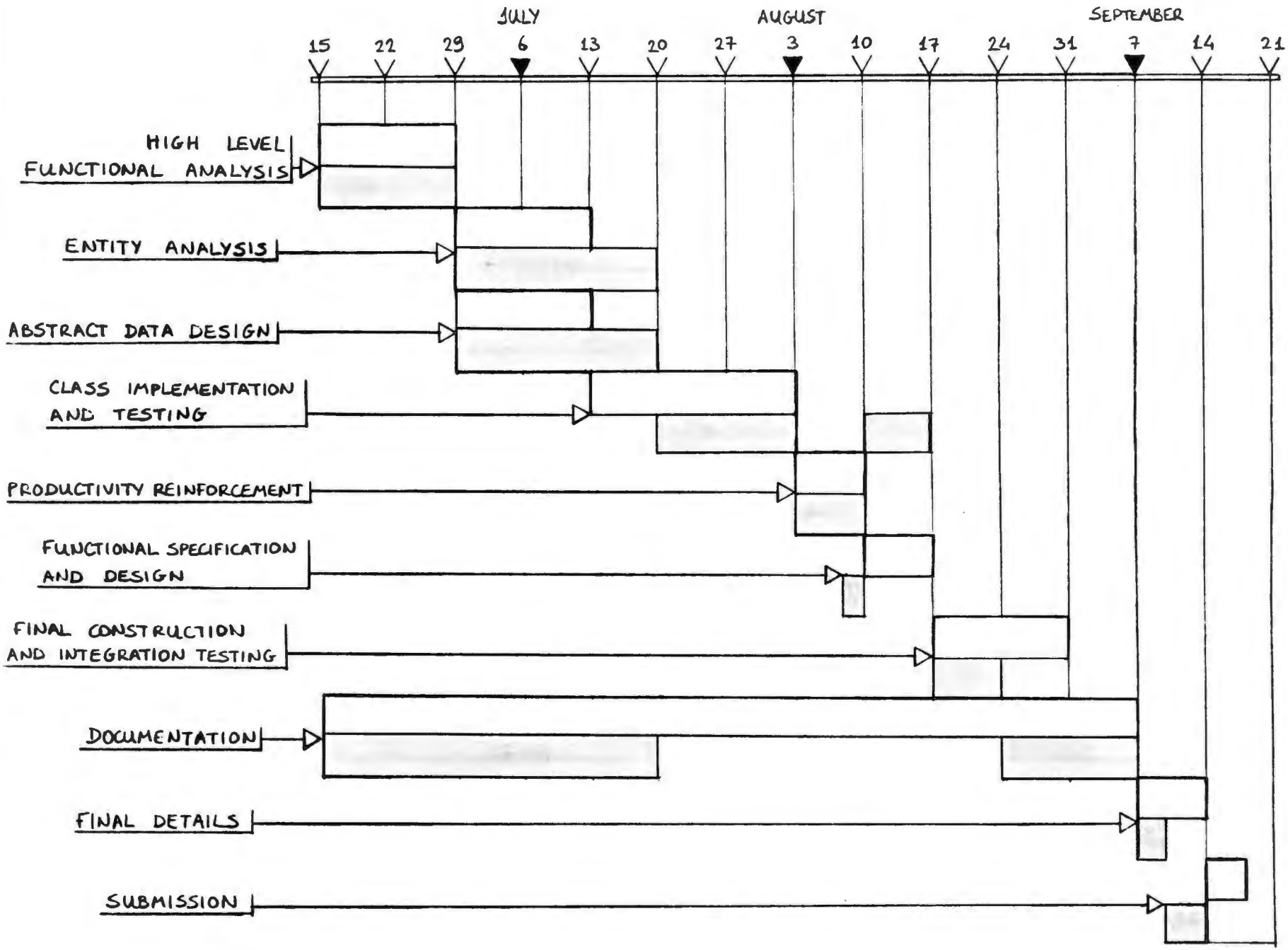
APPENDIX A: Time schedule

How can we break the long tradition of software products that do not keep their time schedules, and even if we did, who would believe that everything went as planned?

So, we present two Gantt charts, one as originally planned, the other showing what actually happened. The pleasant fact is that although the original schedule has been violated, the project has not suffered any severe consequences.

D. Ince[4] decomposes an object oriented development of a system to the following activities, which we use to distribute the available time.

1. *High level functional analysis:* The problem is understood and the requirements specified. This activity took 2 weeks.
2. *Entity analysis:* Discover the entities, actions and attributes of the system.
3. *Abstract data design:* Express the system in terms of abstract data types. Activities 2 and 3 were estimated to take 2 weeks, and took 3 weeks.
4. *Class implementation and testing:* It was estimated to take 3 weeks, took 3 weeks.
5. *Functional specification and design:* Design of the user program.It was estimated to take 1 week, took a few days.
6. *Final construction and integration testing:* Implementation and testing of the program. It was estimated to take 2 weeks, took 1 week.
7. *Documentation:* This was estimated to take 1 week, will take at least another week.



GANTT CHART

ESTIMATED :

ACTUAL :

APPENDIX B: Model of the library

The model of the library can be divided in two parts: The first part recognises the objects that form the library and the relations between them. We use an Entity - Relationship Diagram to show the structure of the library. The second part defines the attributes for each item.

It must be emphasized that this model is not a part of the design of the system, although there is a relation between the model and the design.

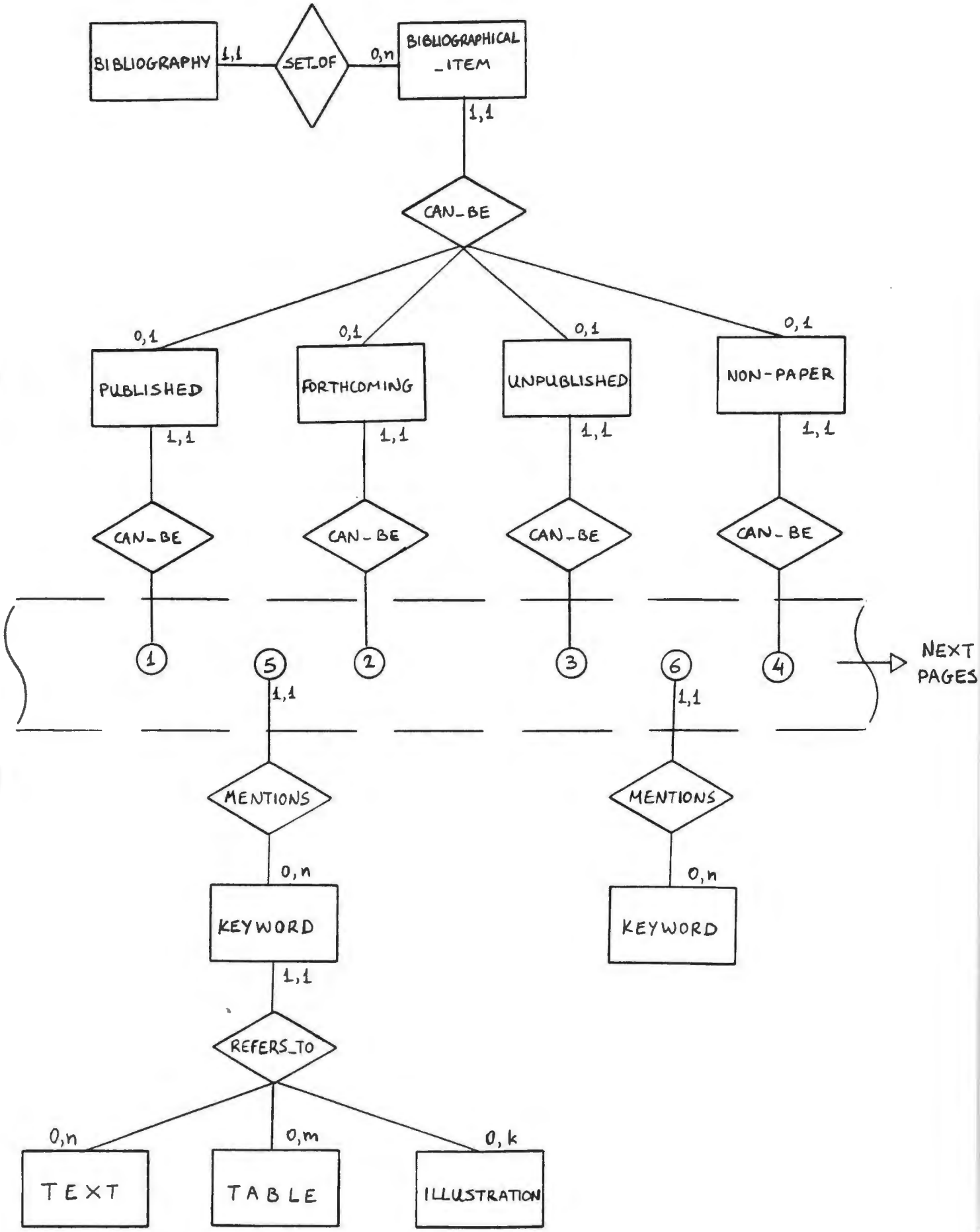
1. Structure of the library

The E-R Diagram starts with a conceptual division of the bibliographical items to published, forthcoming, unpublished and non-paper. The items of each of these types may then consist of other items (ie. a multivolume work consists of volumes, a newspaper of articles etc.); this division corresponds more to a physical division than to a conceptual one. Useful information about the physical structure of the bibliographical items can be found in the "Manual for Writers of Research Papers, Theses and Dissertations"[3] and the Tib manual[2].

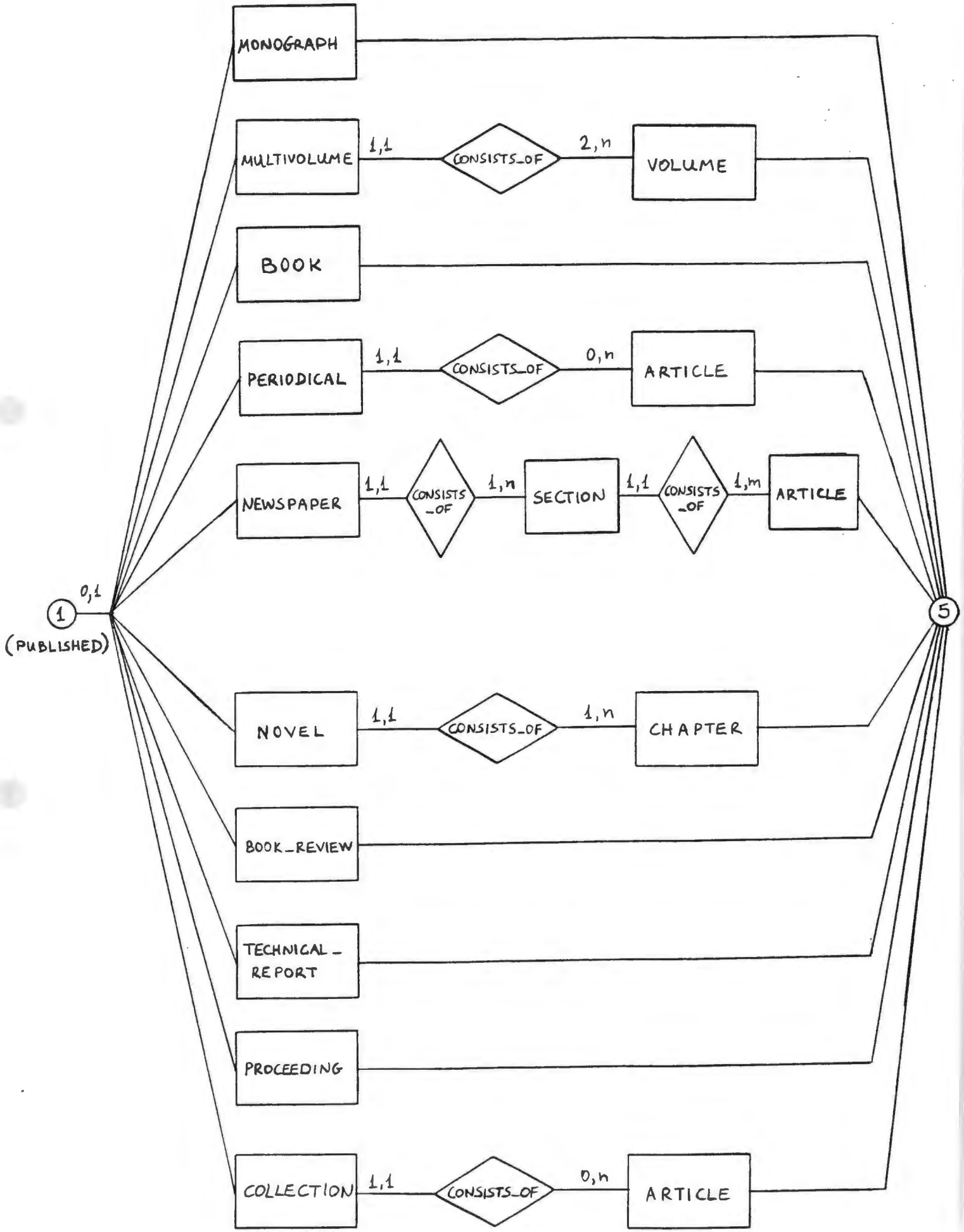
Some of the items mentioned in the former book have not been included in the E-R Diagram, because they were found irrelevant to a computer science research library. These are: plays and long poems, medieval works, scriptural references, and classical references. Series have not been included because they consist of volumes which can be considered as books without loss of information. Encyclopaedias and dictionaries have not been included as it is impractical to enter details for every article they contain, and as they are similar to collections. Novels however, have been included.

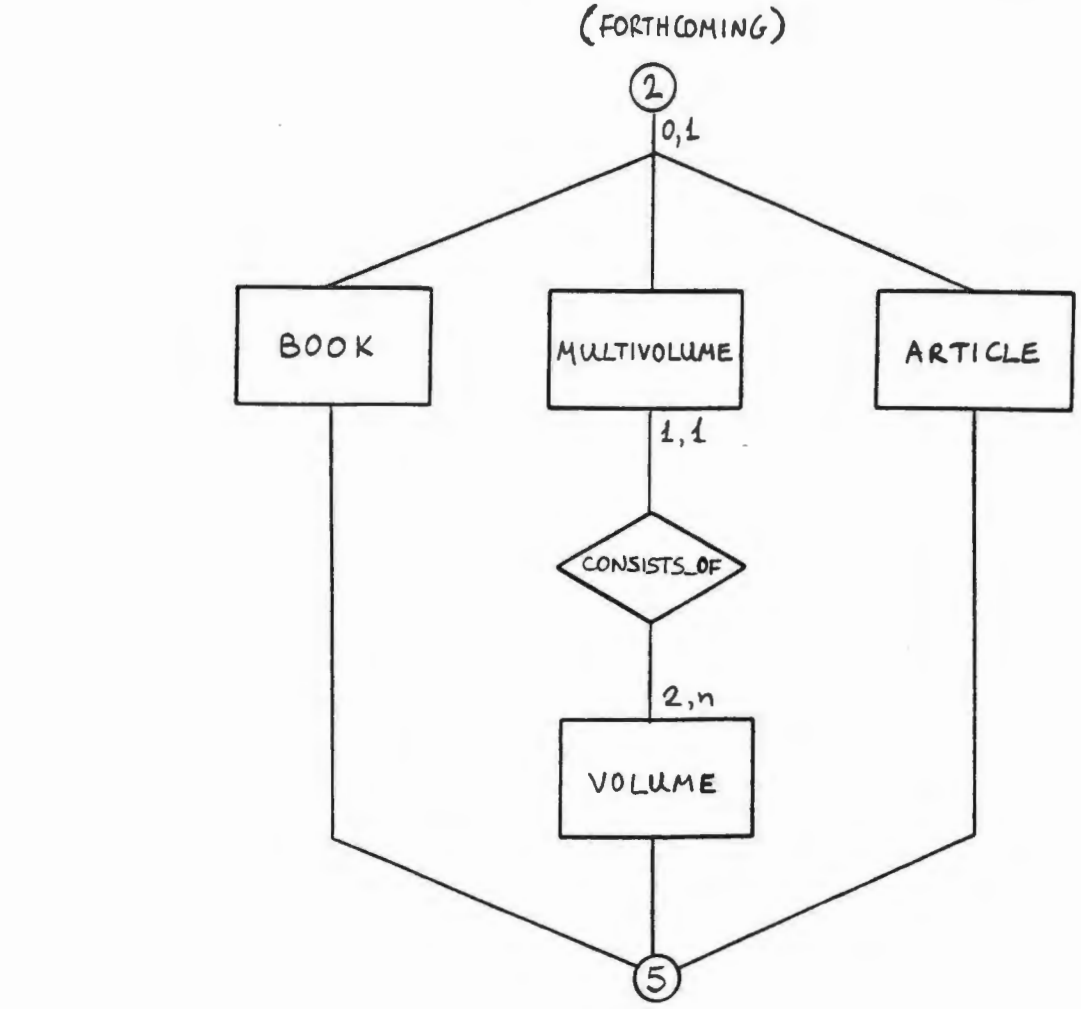
In future more objects might be included. For example, a relatively new kind of published items is the CD ROM which does not belong to any of the above categories. Electronic publishing is another recent way of publishing which could be added to our database structure.

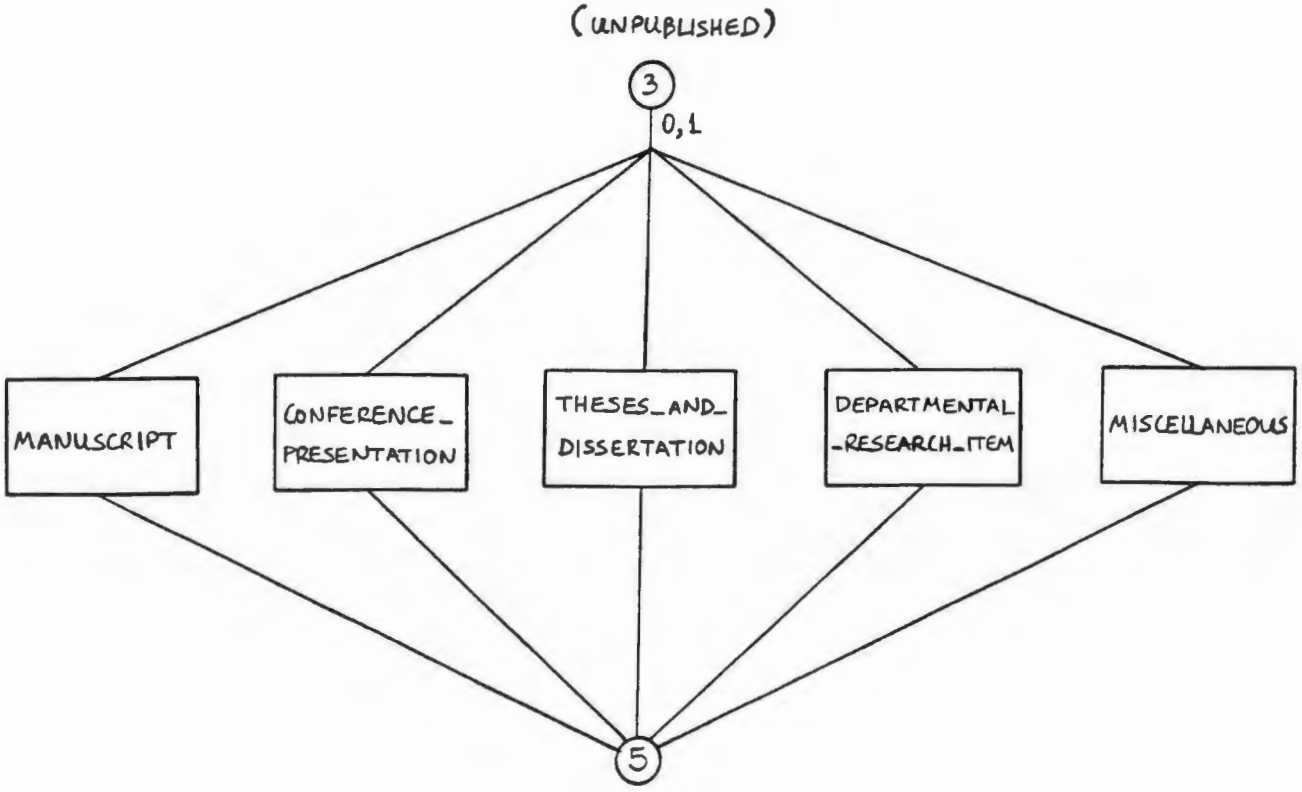
E-R DIAGRAM


BIBLIOGRAPHY
1,1
SET_OF
0,n
BIBLIOGRAPHICAL _ITEM
1,1
CAN_BE
0,1
PUBLISHED
1,1
CAN_BE
1
0,1
FORTHCOMING
1,1
CAN_BE
2
0,1
UNPUBLISHED
1,1
CAN_BE
3
0,1
NON-PAPER
1,1
CAN_BE
4
NEXT PAGES
5
1,1
MENTIONS
0,n
KEYWORD
1,1
REFERS_TO
0,n
TEXT
0,m
TABLE
0,k
ILLUSTRATION
6
1,1
MENTIONS
0,n
KEYWORD

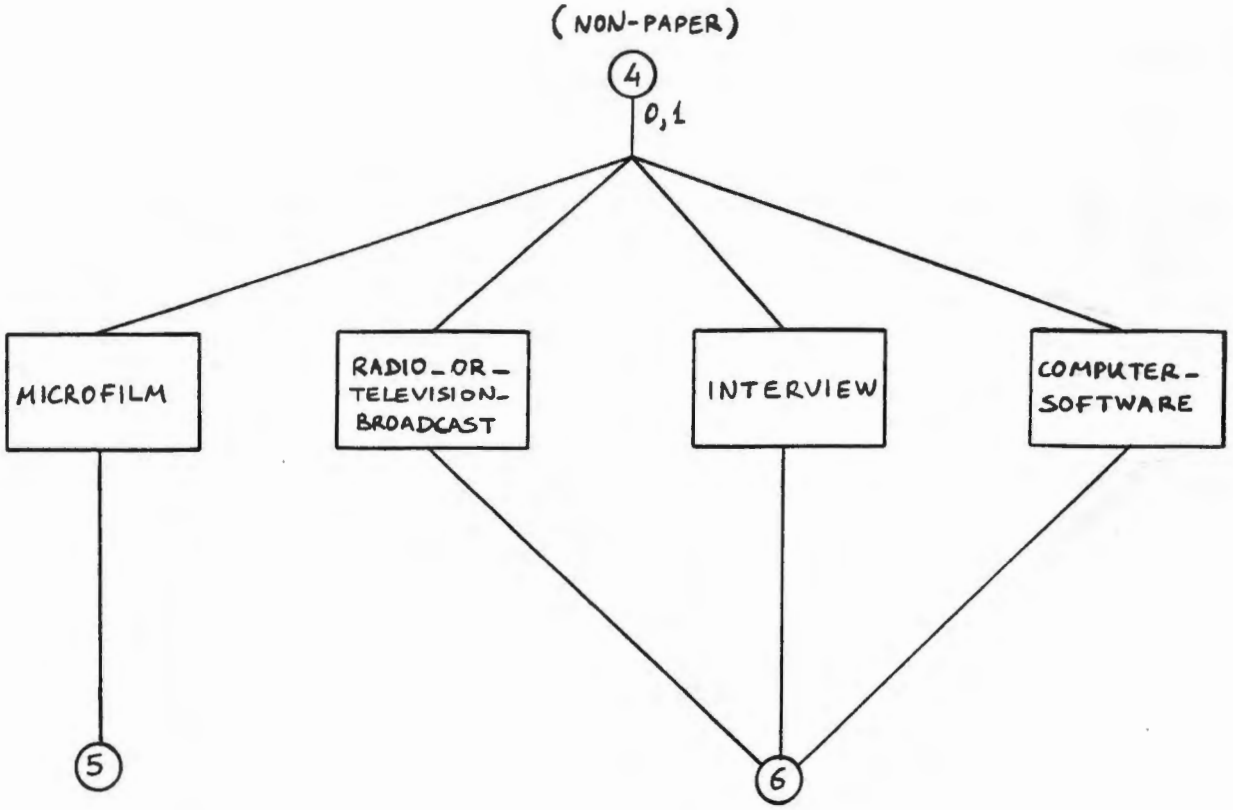
continue


MONOGRAPH
MULTIVOLUME
1,1
CONSISTS_OF
2,n
VOLUME
BOOK
PERIODICAL
1,1
CONSISTS_OF
0,n
ARTICLE
NEWSPAPER
1,1
CONSISTS_OF
1,n
SECTION
1,1
CONSISTS_OF
1,m
ARTICLE
0,1
1
(PUBLISHED)
5
NOVEL
1,1
CONSISTS_OF
1,n
CHAPTER
BOOK_REVIEW
TECHNICAL_REPORT
PROCEEDING
COLLECTION
1,1
CONSISTS_OF
0,n
ARTICLE


(FORTHCOMING)
2
0,1
BOOK
MULTIVOLUME
ARTICLE
1,1
CONSISTS_OF
2,n
VOLUME
5


(UNPUBLISHED)
3
0,1
MANUSCRIPT
CONFERENCE_PRESENTATION
THESES_AND_DISSERTATION
DEPARTMENTAL_RESEARCH_ITEM
MISCELLANEOUS
5

continue


(NON-PAPER)
4
0,1
MICROFILM
RADIO_OR_TELEVISION_BROADCAST
INTERVIEW
COMPUTER_SOFTWARE
5
6

continue

The E-R Diagram then concludes with another conceptual division related to the design: the keywords that refer to the content of an item and can be used for retrieval of items. If the item does not belong to non-paper items, the keywords can refer to tables, illustrations, or simply text, and have page number(s) associated with them.

2. Lists of attributes

The listing of the attributes considers only the items that are physically part of the library. For example, the attributes of a book have been defined , but not the attributes of a published item in general. The attributes of the items have been selected by studying the "Manual for Writers of Research Papers, Theses and Dissertations"[3] and the Tib manual[2].

NOTES:

1. "details of original" are all the attributes of the item involved, except the following fields : authors (including pseudonyms), number of volumes, volume number.

2. "number of pages" refers to the total number of pages of an item, while "page numbers" means the numbers of the first and last page of a part of an item.

1. PUBLISHED ITEMS

- MONOGRAPH : authors (including pseudonyms) - title - publication place - publisher's name - publication date - number of pages - ISBN - abstract

- MULTIVOLUME : authors (including pseudonyms) OR editors OR compilers - title - number of volumes - translator (plus details of original) - publication place - publisher's name - publication date
VOLUME : volume number - authors (including pseudonyms) - title - publication date - number of pages - number of forematter pages - ISBN - abstract

- BOOK : authors (including pseudonyms) OR editors OR compilers - title -

translator (plus details of original) - authors of foreword - edition - publication place - publisher's name - publication date - government number - number of pages - number of forematter pages - ISBN - abstract

- PERIODICAL : title - volume number - publication date - number of pages
  ARTICLE : authors (including pseudonyms) - title - page numbers

- NEWSPAPER : title - publication place - publication date
  SECTION : section name or number
  ARTICLE : authors (including pseudonyms) - title

- BOOK REVIEW : authors (including pseudonyms) - (details of book) - title [of periodical] - publication date - page numbers

- NOVEL : authors (including pseudonyms) - title - edition - publication place - publisher's name - publication date - translator (plus details of original) - number of pages - ISBN - abstract
  CHAPTER : title

- TECHNICAL REPORT : authors - title - publication place - publisher's name - publication date - report number - government number - translator (plus details of original) - number of pages - ISBN - abstract

- PROCEEDING : editors - title - publication place - publisher's name - publication date - conference title - conference place - conference date - number of pages - ISBN - abstract

- COLLECTION : editors - title - publication place - publisher's name - publication date - number of pages - ISBN - abstract
  ARTICLE : authors (including pseudonyms) - title - page numbers

2. FORTHCOMING ITEMS

- MULTIVOLUME : authors (including pseudonyms) OR editors OR compilers - title - number of volumes - publication place - publisher's name

VOLUME : volume number - authors (including pseudonyms) - title - abstract

- BOOK : authors (including pseudonyms) OR editors OR compilers - title - authors of foreword - publication place - publisher's name - abstract

- ARTICLE : authors (including pseudonyms) - title - title of magazine / newspaper

3. UNPUBLISHED ITEMS

- MANUSCRIPT : city of depository - name of depository - name of collection - title of document - abstract

- THESES / DISSERTATION : authors - title - place - institution name - date - degree - abstract

- CONFERENCE PRESENTATION : authors - title - date - number of pages

- MISCELLANEOUS : authors - title - nature of material - position of material - abstract

- DEPARTMENTAL RESEARCH : authors - title - place - institution name - date - report number - government number - number of pages - abstract

4. NON-PAPER ITEMS

- MICROFILM : (book / magazine / newspaper details) - micropublication city - micropublisher's name - micropublication date

- RADIO / TV BROADCAST : title - station name - date - abstract

- INTERVIEW : names - place - date - abstract

- COMPUTER SOFTWARE : product name - release - company name - copyright holder - size - abstract

3. From system model to system design

The system model previously described, has some points that need modification in order for the design to be simple and efficient, without sacrificing its functionality:

- In the real world a volume is part of a multivolume work. However, since each volume may refer to a different - although not completely different - subject, we think that the volume is the conceptual unit that must be emphasised in the design, and the multivolume details could be attributes of the volume. This makes sense if we make the assumption that we are more interested in operations on a volume (like find volumes that refer to a subject) than in operations on multivolume work (like find the titles of the volumes that belong to a multivolume work).

- For the same reasons, articles are considered more important than the periodicals and the newspapers that they belong to, and are treated separately with details of periodicals or newspapers as their attributes. The same holds for collections of articles.

There is an obvious doublication of information, since we repeat some attribute values for many article or volume instances, but when selecting the entities for the design, it seems better to think about different entities as objects with conceptual integrity that talk about specific things, than just as objects that are physically independent and are not part of other objects.

The last approach would permit operations like: list all articles in a specific newspaper. We think however that, if the newspaper is available, this will not be a very useful operation, as it might be simpler to check the newspaper itself. Maybe it would be more useful to concentrate on the efficiency of operations like: find all articles that talk about a specific subject. In this case it is not very simple to check all the newspapers one by one.

APPENDIX C: User manual for the classes

This appendix aims to be a comprehensive guide for using the classes that form the basic structure of the bibliographical database. The first part presents the structure of the system, the second describes the member functions of the classes, and the third shows the interdependencies of the classes.

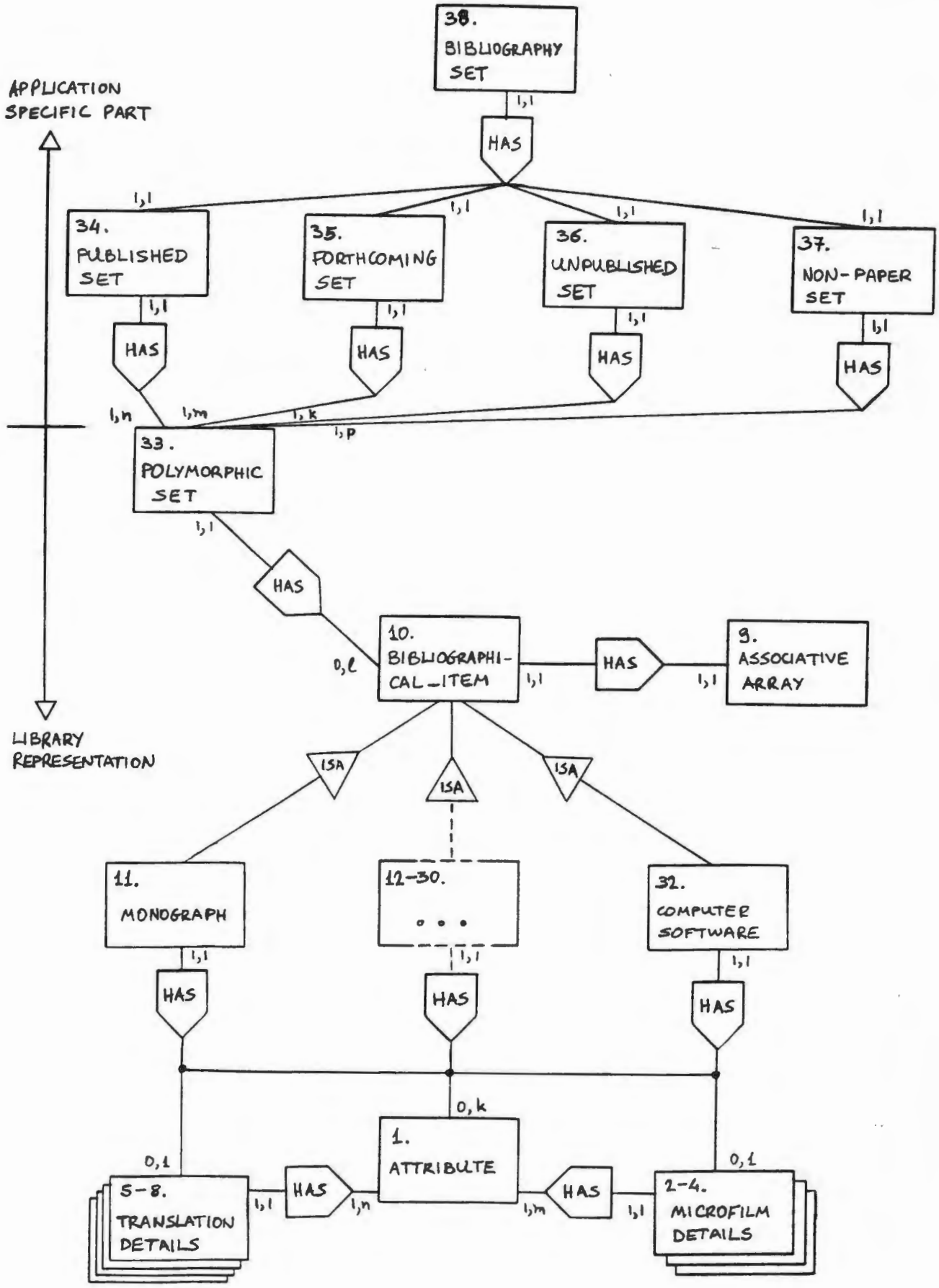
1. Structure of the system

The next figure shows the relations between the objects that form the system. The heart of the system is a collection of sets of various kinds of bibliographical items (set of books, set of newspaper articles, set of microfilms, etc). Each set is an instance of a polymorphic set [33], that contains items of the same kind [11-32] (books, novels, microfilms etc), and may be associated with a file where the items are kept. The polymorphic set will offer facilities for retrieving items by giving a keyword that corresponds to some attributes or by giving a keyword refering to the contents of the item, adding and deleting an item from the set, counting and listing the items in the set.

The polymorphic set will be defined to contain abstract bibliographical items [10] with virtual function declarations: compare an item to another, prompt for the "key" attributes that specify an item etc. Each type of items will inherit this abstract class and define the virtual functions according to its specific attributes.

Some additional virtual functions declared in the bibliographical item class will be: print an item, save or read an item from disk, modify an item, prompt for the attribute values, retrieve the attributes of the item.

Each item will also include an associative array [9] that will be used as an index to store keywords refering to the contents of the item, and page number(s) associated to the keywords that will show where in the item information about a keyword can be found. The associative array will offer operations to check whether a keyword exists and the associated page(s), list all the keywords, add and delete keywords and page number(s).



STRUCTURE OF THE SYSTEM

The abstract bibliography item is inherited by all classes representing the different types of items [11-32]. Each type defines the virtual functions that need redefinition and adds any other functions specific to that type (for example some items are likely to be moved from unpublished to forthcoming or from forthcoming to published). It also declares its attributes as instances of the Attribute [1] class.

The Attribute class contains operations that store and return the value of the attribute, save and read an attribute instance from disk, modify and print the attribute to the standard output, and check whether the value of the attribute is the null string.

The types that can be translated contain a pointer to another class similar to classes [11-32] that keep information about the original item [5-8]. The microfilm type contains pointers to classes representing all the different things a microfilm can hold details for. In our case, a microfilm can hold details about a book, a periodical article, or a newspaper article. This is convenient since we can use pointers to the already defined classes of those items, inside the microfilm class, and we do not need to define additional classes to hold the microfilm details.

The sets of bibliographical items can be grouped together to permit operations that act on many of them at the same time. Each of the published [34], forthcoming [35], unpublished [36], and non-paper [37] set classes will use the corresponding instances of the sets to provide operations such as count, list items and search for a specific item, on the union of the sets that they represent. The bibliography set class [38] will use the previous classes to provide similar operations on all the stored items.

2. Interface of the classes

It is good to be tidy, so the first thing to consider is some rules about storage allocation. There will be a clear separation of responsibilities between the classes and the client program, about the storage allocation: each will be responsible of deleting the storage that it has allocated. When the classes need a structure created by the client program, they will always make a copy of it, and the client program will be responsible for deleting the structure that it has created.

However, there is an exception to this rule: some member functions of the classes will create new objects and pass pointers to these objects to the client program. For example, when you specify an item of the set in order to modify it, the member function of the set that finds the item will make a copy of it and return a pointer to the

copy. In this way the set is protected from implicit modifications. The client program must know which the functions that return new copies are, and delete the structures when it is appropriate.

[1] Attribute

This class has as private data a pointer to a string. The storage for the string is allocated by the class, and the string holds the value of the attribute instance.

INTERFACE

```
const int max_attr_size = 1000;  //maximum size of attribute
const char undefined = '-';      //symbol for undefined attribute

class Attribute {
public:
     Attribute ();
     ~Attribute ();
     void set (char *value); //set the value to "value"
     void get ();            //read from st. input
     void modify ();         //modify from st. input/output
     char *check ();         //examine the value
     void print ();          //print to st. output
     int is_null ();         //see if attribute is null string
     void save_attr (ostream *file);  //save to file
     void read_attr (istream *file);  //read from file
private:
     char *attr_value;
     void get_input (char *str);
};
```

DESCRIPTION

`Attribute ()`

The constructor of the class initialises the pointer 'attr_value' to the null string (" ").

`~Attribute ()`

The destructor de-allocates the storage pointed by 'attr_value'.

```
void set (char *value)
```

This function deletes the string that is currently pointed by 'attr_value'. If 'value' is 0 or it points to the 'undefined' character, the 'attr_value' is set to the null string, else a copy of 'value' is made and 'attr_value' points to that copy.

```
void get ()
```

This function reads the attribute value from the standard input. It uses the function 'set' to store the value. The string cannot be greater than* The end of the string is considered to be the newline character. The standard input must be clear when this function is called, else the function will include the contents of 'cin' in the attribute value. Usually after the user has typed a command, he/she presses 'return'. This must be removed from 'cin' before calling this function. The function will leave the stream clear.

```
void modify ()
```

This function prints the attribute value to the standard output, changes line, and waits for the user to enter a new value to the standard input. If the user presses 'return', the value remains the same. If the user enters the 'undefined' character, the value becomes the null string. Any other input will cause the value to become the string of the input. The string cannot be greater than 'max_attr_size' characters. The end of the string is considered to be the newline character. The standard input must be clear when this function is called, else the function will include the contents of Usually after the user has typed a command, he/she presses 'return'. This must be removed from 'cin' before calling this function. The function will leave the stream clear.

```
char *check ()
```

Returns the pointer 'attr_value' that points to the current value.

```
void print ()
```

Prints to the standard output the value for the Attribute instance. If the value is the null string, it prints the 'undefined' character.

*"max_attr_size".

```
int is_null ()
```

Returns true if the value is the null string.

```
void save_attr (ostream *file)
```

Writes the value to the stream pointed by 'file'. This stream must be associated with a file. The function writes a space character first, then the length of the string, another space, the string itself, and another space. The space after the length is important because it permits the 'read' function to consider it as an integer.

```
void read_attr (istream *file)
```

Deletes previous value and reads the new value from the stream 'file' which must be associated to a file.

[2-4] Microfilm details

A microfilm can be about a book, a periodical article, or a newspaper article. We already have classes about these items [13-14-15], so the Microfilm class [29] will contain in the private part pointers to all of these classes that hold the necessary details, plus the attributes that are specific to microfilms. Two of the three pointers will always be 0 and only one will point to a class instance, depending on the content of the microfilm instance.

[5-8] Translation details

These are classes that hold details about the original when the item is translated. The items that can be translated are: volume, book, novel, and technical report. Except from the appropriate attributes, they have functions like 'prompt', 'modify', 'print_attribute', 'copy_item', 'save' and 'read', similar to the corresponding functions of the item types (see Bibliog_item [10]. The items that can be translated will contain a pointer to the relevant translation details class in their private part, and use the functions listed above in their own similar functions.

INTERFACE

An example is the Transl_book class:

```
class Transl_book {
public:
     Transl_book ();
     ~Transl_book ();
     void prompt ();        //from st. input
     void modify ();        //from st. input
     void print_attrib (ostream *out);
     Transl_book *copy_item ();        //returns identical copy
     void save_item (ostream *file);
     void read_item (istream *file);
private:
     Attribute *title;
     Attribute *edition;
     Attribute *publ_place;
     Attribute *publ_name;
     Attribute *publ_date;
     Attribute *no_of_pages;
     Attribute *no_of_fore_pages;  //number of forematter pages
     Attribute *governm_num;       //government number
     Attribute *isbn;
};
```

DESCRIPTION

Same as Bibliog_item class [10].

[9] Associative array

An associative array of strings. The strings are copied in newly created storage when they are inserted in the array.

INTERFACE

```
const int max_element_num = 50;  //max num of elements

class Assoc_array {
public:
     Assoc_array ();
     ~Assoc_array ();
     void put (char *keyword, char *value); //add-update entry
     const char *at (char *keyword);        //examine value
```

```
     int cancel (char *keyword);            //delete entry
     void print_index (ostream *out);       //print contents
     Assoc_array *copy_index ();      //make a copy of hash table
     int count ();                    //count entries
     void save_index (ostream *out);  //save to file
     void read_index (istream *in);   //read from file
private:
     int element_num;   //current number of elements
     Index_entry *hash [max_size];
     int hash_number (char *string); //the hashing function
     int position (char *keyword);   //the collision strategy
};
```

DESCRIPTION

`Assoc_array ()`

Initialises private variables.

`~Assoc_array ()`

De-allocates memory space.

`void put (char *keyword, char *value)`

If 'keyword' does not exist in the array, and the number of elements is less than 'max_element_num', the pair 'keyword' - 'value' is added to the array. If number of elements is equal to 'max_element_num', a warning message is printed to 'cerr'. If 'keyword' exists in the array, the previous value that corresponds to that keyword is substituted by the 'value' of the parameter. The array does not distinguish between small and capital letters when searching for a keyword. However, the keyword will be kept as it was the last time the function 'put' was used for it.

`const char *at (char *keyword)`

If 'keyword' exists in the array, the corresponding value is returned, else 0 is returned. The function is declared 'const' because the user must not use this function to change the value. The type of letters of 'keyword' is not important.

```
int cancel (char *keyword)
```

If 'keyword' does not exist, 0 is returned, else the pair keyword - value corresponding to 'keyword' is deleted from the array and 1 is returned. The type of letters of 'keyword' is not important.

```
void print_index (ostream *out)
```

It prints to the stream 'out' all the pairs in the array, arranged in two columns: keywords to the left, values to the right. The distance between the columns is set to 40 characters.

```
Assoc_array *copy_index ()
```

Returns a pointer to a copy of the current Assoc_array instance. The copy contains the same entries as the original and can be used independently.

```
int count ()
```

Returns the number of the pairs keyword - value in the array.

```
void save_index (ostream *out)
```

Writes the contents of the array to the stream 'out' which must be connected to a file. This is done as follows: a space character is written, then the number of pairs of the array, then a space, and then the function enters a loop. For all the pairs: write a space, write the length of the keyword, a space, the keyword itself, a space, the length of the string, a space, the string itself, another space. Only the spaces after the numbers are important, as they permit to the read function to treat the numbers as integers.

```
void read_index (istream *in)
```

Reads stream 'in' which must be connected to a file, and loads the associative array. If the array has already some elements, it will keep them plus the new elements read from the stream.

[10] Bibliog_item

It is an abstract class that consists of declarations of virtual functions, a pointer to Assoc_array [9] and a pointer to Bibliog_item. All the bibliographical items [11-32] will inherit this class (and the pointers) and will define the abstract virtual functions.

INTERFACE

```
class Bibliog_item {
public:
     virtual void prompt () = 0;
     virtual void prompt_for_key () = 0;
     virtual void modify () = 0;
     virtual void print_attrib (ostream *out) = 0;
     virtual void print_key () = 0;
     virtual Bibliog_item *copy_item () = 0;
     virtual int compare (Bibliog_item *item) = 0;
     virtual int compare_attrib (char *keyword) = 0;
     virtual void save_item (ostream *file) = 0;
     virtual void read_item (istream *file) = 0;

     Assoc_array *index;  //all bibl. item kinds have index
     Bibliog_item *next_item; //and can be linked in a list
};
```

DESCRIPTION

The following descriptions refer to the definition of the above functions given in the classes of the various bibliographical item types [11-32] that inherit the Bibliog_item class.

`void prompt ()`

Uses the function 'get' of the Attribute class [1] to take the values of the item attributes. For all the attributes of the item, it prints the name of each attribute to the standard output and calls the 'get' function of the attribute instance. Use the same precautions, as described in the 'get' function.

`void prompt_for_key ()`

Exactly as the function 'prompt', but only for the key attributes.

```
void modify ()
```

The key functions are printed to the standard output by calling the function Then for each of the rest of the attributes, the attribute name is printed and the attribute value is modified by calling the 'Attribute :: modify' function of the attribute instance. Use same precautions as for

```
void print_attrib (ostream *out)
```

For each attribute, the attribute name and the attribute value is written to the stream 'out', only if the attribute value is not the null string. If the value is null, this attribute is omited.

```
void print_key ()
```

Prints the names and the values of the key attributes to the standard output. It uses the 'print' function of the Attribute class [1], that means that if an attribute value is null, the attribute name and the 'undefined' character (see Attribute class [1]) are printed.

```
Bibliog_item *copy_item ()
```

It creates a new item identical to the current instance, by using the 'set' and 'check' functions of the attribute class [1] to copy the attributes, and the 'copy_index' function of the Assoc_array class [9] to copy the associative array. It then returns a pointer to the copy, which can be used independently of the original.

```
int compare (Bibliog_item *item)
```

This function compares two bibliographical item instances: the current instance and the instance pointed by the parameter 'item'. The two items must be of the same bibliographical item type. The comparison involves all the key attributes. Some attributes are considered more important than others and are compared first. If those attributes are equal, the next most important attributes are compared. If all the key attributes are equal, the two items are considered equal and 0 is returned. If an attribute of the current item is alphabetically greater than the corresponding attribute instance of the parameter, the current item is considered greater and 1 is returned. In a similar

way -1 is returned if the current item is less than the parameter. Note that when comparing the attributes the type of letters (small - capital) is not important.

```
int compare_attrib (char *keyword)
```

A subset of attributes of the item is defined. If 'keyword' is part of any of them, 1 is returned, else 0 is returned. The keyword need not be the complete attribute value, but only part of it. The type of letters is not important. The attributes are chosen according to whether they are useful for retrieval purposes.

```
void save_item (ostream *file)
```

Saves the item to disk by using the 'save_attr' function of the Attribute class [1] for all the attribute instances, and the 'save_index' function of the Assoc_array class [9]. The stream 'file' must be connected to a file. For the items that may have authors or compilers or editors, the function will write to disk a letter (between spaces) indicating the type of the creator. The same technique is used in the Microfilm class to save the type of the Microfilm.

```
void read_item (istream *file)
```

Reads the item from disk by using the 'read_attr' function of the Attribute class [1] for all the attribute instances, and the 'read_index' function of the Assoc_array class [9]. The stream 'file' must be connected to a file.

[11] Monograph
[12] Volume
[13] Book
[14] Periodical article
[15] Newspaper article
[16] Book review
[17] Novel
[18] Technical report
[19] Proceeding
[20] Collection article

[21] Forthcoming volume
[22] Forthcoming book
[23] Forthcoming article
[24] Manuscript
[25] Theses
[26] Conference presentation
[27] Miscellaneous
[28] Research
[29] Microfilm
[30] Broadcast
[31] Interview
[32] Software

These classes inherit the Bibliog_item class, define the virtual functions according to their specific attributes, and declare various Attribute instances as private data. Item types that may have authors, compilers or editors, include a private variable denoting whether the relevant attributes refer to authors, compilers or editors. Items that can be translations (Volume, Book, Novel, Technical_report) keep a pointer to a class holding details about the original. A microfilm can be about a book, a periodical article, or a newspaper article, so the Microfilm class keeps pointers to the relevant classes.

Not all the above classes have been implemented. The ones that have, are:
[11] Monograph
[13] Book
[14] Periodical article
[22] Forthcoming book
[23] Forthcoming article
[25] Theses
[28] Research
[29] Microfilm
[32] Software.

INTERFACES

Similar to the Bibliog_item class [10].

DESCRIPTION

Similar to the Bibliog_item class [10].

Some classes whose instances can be transfered from unpublished to forthcoming or forthcoming to published, will include functions like:

```
FArticle *Research :: make_forthcoming ()
```

Create the corresponding forthcoming item, copy all the values of the common attributes to the new item by using the 'set' and 'check' functions of the Attribute class [1], prompt for the rest of the forthcoming item attributes by using the 'get' function of the Attribute class, and return a pointer to the new forthcoming item. These functions are declared 'friend' of the corresponding forthcoming class. See precautions of 'get' function. It is the responsibility of the user program to add the new forthcoming item to the corresponding set, and to delete (if necessary) the unpublished item from its set.

```
Book *FBook :: publish_fbook ()
Period_article *FArticle :: farticle_in_period ()
```

Create the corresponding published item, copy all the values of the common attributes to the new item by using the 'set' and 'check' functions of the Attribute class [1], prompt for the rest of the published item attributes by using the 'get' function of the Attribute class, and return a pointer to the new published item. These functions are declared 'friend' of the corresponding published class. See precautions of 'get' function. It is the responsibility of the user program to add the new published item to the corresponding set, and to delete (if necessary) the forthcoming item from its set.

[33] Polyset

A polymorphic set defined to contain abstract bibliographical items, by using the declaration of the Bibliog_item class [10]. Each instance of the set can be connected to a file, or not. Different instances of the set can be used to hold instances of different types of bibliographical items (classes 11 - 32).

INTERFACE

```
const int initial_max_size = 100;  //start with this max no of items
const int size_step = 50;          //increase it if not adequate

class Polyset {
public:
     Polyset (char *filename = 0);  //default is: not file
     ~Polyset ();
     int add (Bibliog_item *item);
     int replace (Bibliog_item *item); //for another with same key
     int remove (Bibliog_item *item);
     Bibliog_item *specify (Bibliog_item *item);  //find item
     int count ();
     Bibliog_item *list ();     //all items
     Bibliog_item *retrieve_by_attrib (char *keyword);
     Bibliog_item *retrieve_by_context (char *keyword);
     void save_set ();
     void read_set (Bibliog_item *item);
private:
     char *file;
     int max_size;  //max size of set can change
     int element_number;
     Bibliog_item **items;
     int set_has_changed;  //if yes we need to save it to file
     int search (Bibliog_item *item);  //binary search
     void increase_size ();  //if current max size inadequate
};
```

DESCRIPTION

`Polyset (char *filename = 0)`

Initialises variables. Two variables of particular importance are the If the Polyset instance is declared with a file name, then a copy of the parameter be 0. The 'set_has_changed' is used to show if the set must be saved to the file (if it is true, the destructor must overwrite the file with the new contents before deleting them). This variable is not initialised to 0 but to 1, because when the set is created we do not know if the contents of the set (which is empty) corresponds to the contents of the file (which may not be empty). Be careful: if you delete a newly created set you loose the associated file.

`~Polyset ()`

If the set is associated to a file and the contents of the set have changed since the last

'read_set' command, the destructor will save the contents by overwriting the corresponding file. The destructor also de-allocates the storage.

```
int add (Bibliog_item *item)
```

Uses the function 'compare' of the Bibliog_item class [10] to see if an item equal to the one pointed by 'item' exists in the set. If it does, 0 is returned. If it does not exist, a copy of the item is made by calling the The copy is added to the set and 1 is returned.

```
int replace (Bibliog_item *item)
```

This function uses the function 'compare' of the Bibliog_item class [10] to see if an item equal to the one pointed by 'item' exists in the set. If it does not, 0 is returned. If it exists, this item is deleted from the set, and a copy of the item pointed by the parameter 'item' is made by calling the 'copy_item' function of the Bibliog_item class [10]. The copy is added to the set and 1 is returned. Remember that the item that has been deleted and the one that has been inserted into the set, are equal if they have the same key attribute values, but the rest of the attribute values may not be the same.

```
int remove (Bibliog_item *item)
```

Uses the function 'compare' of the Bibliog_item class [10] to see if an item equal to the one pointed by 'item' exists in the set. If it does, the item is deleted from the set and 1 is returned. If it does not exist, 0 is returned.

```
Bibliog_item *specify (Bibliog_item *item)
```

Uses the function 'compare' of the Bibliog_item class [10] to see if an item equal to the one pointed by 'item' exists in the set. If it does, a copy of this item is made by using the 'copy_item' function of the Bibliog_item class [10] and a pointer to this copy is returned. If it does not exist, 0 is returned.

```
int count ()
```

Returns the number of the elements in the set.

```
Bibliog_item *list ()
```

If the set is empty, 0 is returned. If the set is not empty, a pointer to the head of a linked list containing all the elements of the set is returned.

```
Bibliog_item *retrieve_by_attrib (char *keyword)
```

Uses the function 'compare_attrib' of the Bibliog_item class [10] to find the items of the set that contain 'keyword' in a specific subset of their attributes. If none of the items satisfy the condition, 0 is returned, else a pointer to the head of a linked list containing all the items of the set that satisfy this condition is returned.

```
Bibliog_item *retrieve_by_context (char *keyword)
```

Uses the function 'at' of the Assoc_array class [9] to find the items of the set that contain 'keyword' in their associative array. If none of the items satisfy the condition, 0 is returned, else a pointer to the head of a linked list containing all the items of the set that satisfy this condition is returned.

```
void save_set ()
```

Opens the associated file, and overwrites it with the current contents of the set. If the file does not exist, it is created. The function writes a space character, the number of the elements of the set, another space, and then calls the 'save_item' function of the Bibliog_item class [10] for all the items in the set. The second space is important because it permits to the 'read_set' function to read the number of the elements as an integer. If this function is called and the set instance is not declared with a file name, an error message will appear.

```
void read_set (Bibliog_item *item)
```

This function opens the file and reads the number of the elements of the set. It then

uses the parameter 'item' by applying to it the function number of the elements. Each time the function 'read_item' is used, a copy of the item that has been read is added to the set by using the 'add' function of the Polyset class. The item type of the instance that 'item' points to, must be the same as the item type of the instances written to the file. If this function is called and the set instance is not declared with a file name, an error message will appear.

[34] Published
[35] Forthcoming
[36] Unpublished
[37] Non-paper

Sets that provide aggregate operations on different instances of the Polyset class [33] holding various types of bibliographical items. Pointers to the Polyset instances are received as constructor parameters. All these sets offer the same functions, the only difference is their constructor parameters.

INTERFACES

```
class Published {
public:
     Published (Polyset *m, Polyset *pa, Polyset *b);
     int count ();
     Bibliog_item *list ();
     Bibliog_item *retrieve_by_attrib (char *keyword);
     Bibliog_item *retrieve_by_context (char *keyword);
private:
     int member_num;
     Polyset *monographs;
     Polyset *period_articles;
     Polyset *books;
     Bibliog_item *link_lists (Bibliog_item **heads);
     Bibliog_item *tail (Bibliog_item *head);
};

class Forthcoming {
public:
     Forthcoming (Polyset *a, Polyset *b);
     int count ();
     Bibliog_item *list ();
     Bibliog_item *retrieve_by_attrib (char *keyword);
     Bibliog_item *retrieve_by_context (char *keyword);
private:
     int member_num;
     Polyset *farticles;
```

```
        Polyset *fbooks;
        Bibliog_item *link_lists (Bibliog_item **heads);
        Bibliog_item *tail (Bibliog_item *head);
};

class Unpublished {
public:
        Unpublished (Polyset *r, Polyset *t);
        int count ();
        Bibliog_item *list ();
        Bibliog_item *retrieve_by_attrib (char *keyword);
        Bibliog_item *retrieve_by_context (char *keyword);
private:
        int member_num;
        Polyset *research_set;
        Polyset *theses_set;
        Bibliog_item *link_lists (Bibliog_item **heads);
        Bibliog_item *tail (Bibliog_item *head);
};

class Non_paper {
public:
        Non_paper (Polyset *s, Polyset *m);
        int count ();
        Bibliog_item *list ();
        Bibliog_item *retrieve_by_attrib (char *keyword);
        Bibliog_item *retrieve_by_context (char *keyword);
private:
        int member_num;
        Polyset *soft_products;
        Polyset *microfilms;
        Bibliog_item *link_lists (Bibliog_item **heads);
        Bibliog_item *tail (Bibliog_item *head);
};
```

DESCRIPTION

```
Published (Polyset *m, Polyset *pa, Polyset *b)
Forthcoming (Polyset *a, Polyset *b)
Unpublished (Polyset *r, Polyset *t)
Non_paper (Polyset *s, Polyset *m)
```

The constructors of the sets receive pointers to Polyset class [33] instances that contain different types of bibliographical items (classes [11-32]), and enable the sets to use the facilities of the Polyset instances in order to provide the following functions:

```
int count ()
```

It uses the 'count' function of the Polyset class [33] to retrieve the number of items of

each Polyset instance. Then it adds the numbers and returns the number of the items of the set.

```
Bibliog_item *list ()
```

It uses the 'list' function of the Polyset class [33] for all the Polyset instances. Then it links the lists together and returns a pointer to the head of a list containing all items in the set.

```
Bibliog_item *retrieve_by_attrib (char *keyword)
```

It uses the 'retrieve_by_attrib' function of the Polyset class [33] for all the Polyset instances. Then it links the lists together and returns a pointer to the head of a list of all the items in the set, whose a subset of their attributes contain the string 'keyword'.

```
Bibliog_item *retrieve_by_context (char *keyword)
```

It uses the 'retrieve_by_context' function of the Polyset class [33] for all the Polyset instances. Then it links the lists together and returns a pointer to the head of a list of all the items in the set, whose the associative array (class [9]) contains the string 'keyword'.

[38] Bibliography

Will contain pointers to instances of classes 34-37 and provide the same operations as those classes, but on all the bibliographical items that exist in the database.

Auxiliary classes

There are two classes used to support member functions of the classes described above. These are the classes 'error' and 'auxiliary'.

INTERFACES

Class 'error':

```
//print an error message and stop the program
void error (char *message);
```

Class 'auxiliary':

```
//converts a string to small letters
char *low_case (char *initial_str);
//see if 'key' is part of 'word'
int match (char *word, char *key);
```

DESCRIPTION

```
void error (char *message)
```

Prints the 'message' to cerr and causes the program to stop. Used mainly when there is a storage allocation failure.

```
char *low_case (char *initial_str)
```

Returns a pointer to a new string that contains the lower-case version of the string 'initial_str'. Used by the bibliographical item types [11-32].

```
int match (char *word, char *key)
```

Searches through the string 'word' to find if 'key' is part of 'word'. If it is 1 is returned, else 0 is returned. Used by the bibliographical item types [11-32].

3. Interdependencies of the classes

The next figure shows the classes that have been implemented as part of the basic system. An arrow from one class to another means that the first class includes the second.

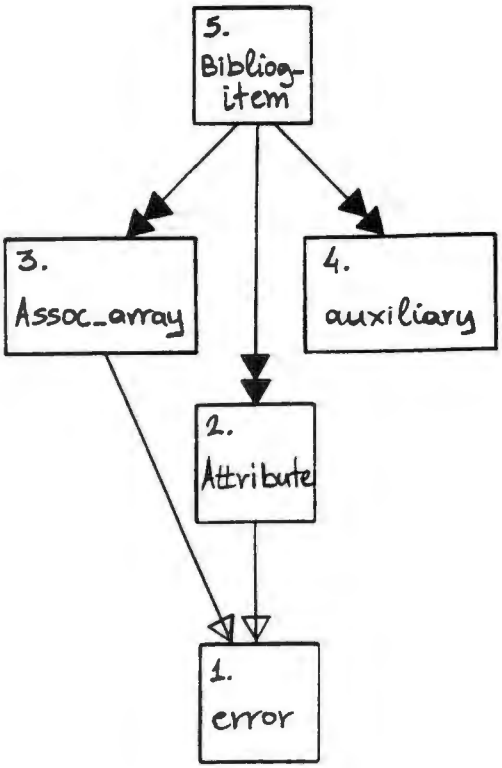
A class may include another class because it uses facilities that the second class offers, and the first class cannot work or be compiled without the second. For example, the Bibliography class includes the Published class because it uses the member functions of the Published class. This kind of relationship is represented by a single white arrow.

A class may also include another because the second class conceptually belongs there. For example the Bibliog_item class includes the Assoc_array class, the Attribute class, and the auxiliary class, although it does not use any of them. However all the items that inherit the Bibliog_item class use all three classes, so it seems convenient and semantically right for these classes to be included in the Bibliog_item class. Another example is the class Forthcoming that includes all the forthcoming item classes, although it does not use them. The fact that it includes them can make a user program simpler and clearer. This relationship is represented by a double dark arrow.

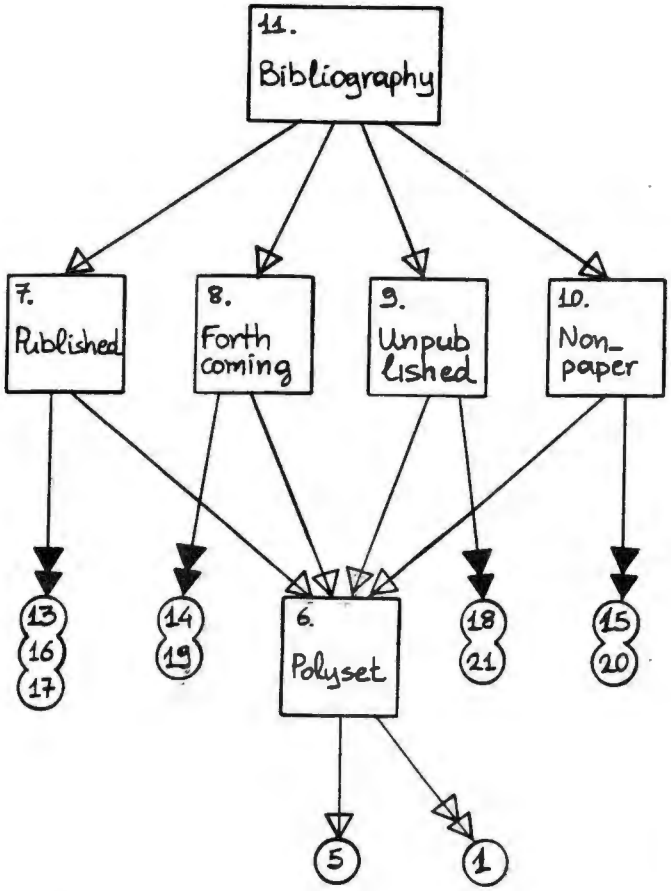
Another case is when a class uses a second class but does not need to include it, because it is implicitly included already. This is the case of the class Polyset that uses the class error, which is implicitly included because the class Bibliog_item includes "error" and is included in "Polyset". However all this is not very clear, so it seemed to us conceptually better to explicitly include the error class in the "Polyset" again. This relationship, where a class is used and is implicitly included but is also included explicitly a second time for reasons of clarity, is represented by a double white arrow.

Complex inclusion relationships may lead to redefinitions of classes. To avoid this, we find the classes that are likely to be redefined or that are included by more than one classes, and we protect them against redefinitions. The classes protected in this way are shown like squares, the rest of the classes like rectangulars.

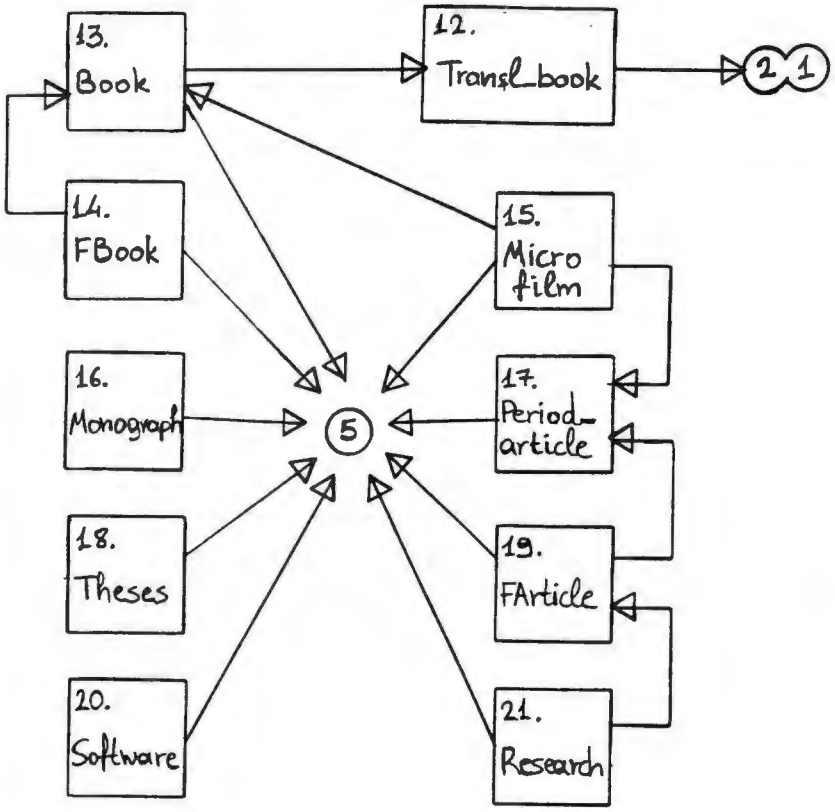
a. Basic structure



b. Sets structure



c. Items structure

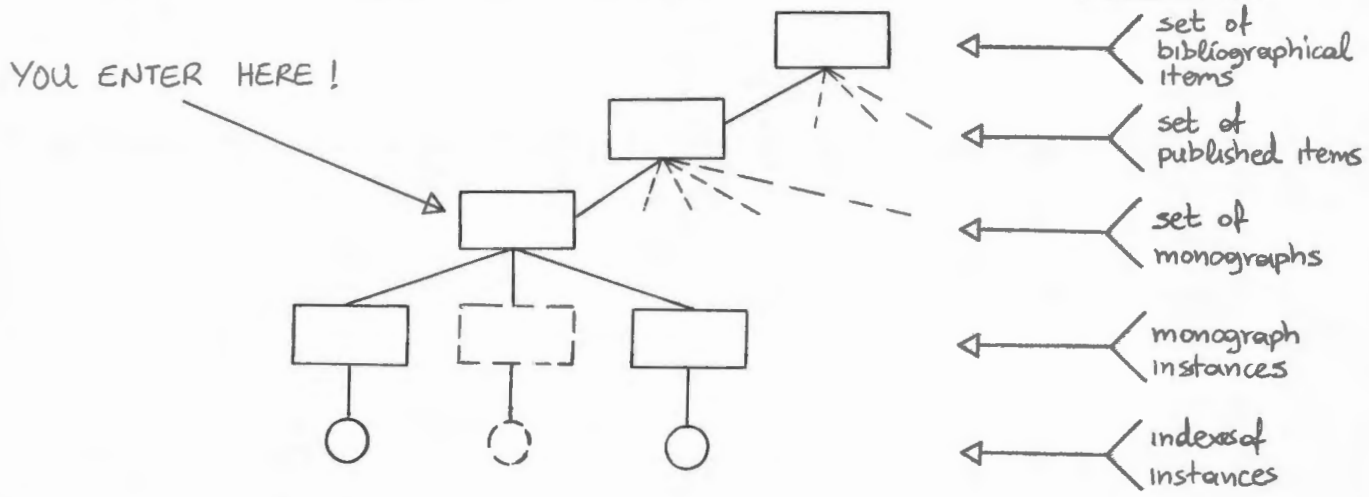


Interdependencies of the classes

BIBLIO

RUN THE PROGRAM: % cd ~ystavrak/last_proj/code ↵
% biblio ↵

FILE WITH DATA IS CALLED: data (same directory)



OPERATIONS

set level:

i		insert new item in the set
d		delete item from set
n		number of items in set
p		print all items (only key attributes) in set
a	<keyword>	find items with "keyword" as part of authors, pseudonyms, title, publisher.
c	<keyword>	find items with "keyword" in their index.
w	<filename>	write set to "filename"
r	<filename>	read set from "filename"
s		specify an item and change level to item
q		quit - exit program.

item instance level:

p		print item
m		modify item attributes (except key attributes)
r		replace original item in set with current item (modified).
i		change level to index.
q		change level to set.

index level:

i	<key> <value>	insert pair (if "value"="null", it will be left empty)
a	<key>	see the value of the "key"
c	<key>	cancel a pair
p		print index
n		number of pairs in index
q		change level to item instance

PS. Comments about bugs or changes ~~ab~~ the code welcome before Monday 24, when I stop implementing and start writing up the report (e-mail me).