An Object-Oriented Software Model for Students' Registration and Examination Result Processing in Nigerian Tertiary Institutions

***Bamigbola, O.M. [1] , Olugbara, O.O. [2] and Daramola, J.O.** [3]

1, 3: Department of Mathematics/Computer Science, University of Ilorin, Ilorin, Nigeria

2: Department of Computer and Information Technology, Covenant University, Otta, Nigeria.

* Author to Correspond on paper

ABSTRACT

The principles of Object-Oriented Software Engineering are employed to model a software application known as Undergraduate Registration and Examination Processing System (SPERU) for Nigerian Tertiary institutions. Rapid Application Development (RAD) tools are utilized in its implementation to achieve an excellent software system.

SPERU is herein presented , as a typical instance of a well modelled software system, that testifies to the beauty, power and supremacy of the Object-Oriented paradigm

The result of applying software predictor metrics indicates that SPERU is reliable and elegant.

1 Introduction

The effort expended in the process of registration of students and computation of their examination results is awesome. Quite worrisome is the fact that these processes are carried out every academic session, putting the operators in a continuous and ever demanding cycle. The computation of examination results and registration of students is obviously an object-centered activity, the student being the dominant object in this case. Hence, the need to evolve not just a computerized process, but an object-oriented software design and implementation that will effectively and efficiently capture all the important objects associated with the registration and examination result processing within the University and the interactions among the objects.

This genuine and noble desire necessitated the design and development of the Undergraduate Registration and Examination Processing System (SPERU) software.

2. Object–Oriented Software Design

Object-oriented design is a design strategy based on abstraction, encapsulation and polymorphism. It views a software system as a set of interacting objects with their own private state. This

makes it different from the functional design which views software system as a set of functions. Object-oriented design has been gaining good publicity and acceptance since the late 1980s.

Apart from the business systems domain, it has become the predominant design strategy for new software systems. The characteristics of an Object-Oriented Design (OOD) are:

(1) Objects are abstractions of real-world or system entities, which are responsible for managing their own private state and offering services to other objects.

(2) Objects are independent entities that may readily be changed because state and representation information are held within the object. Changes to the representation may be made without reference to other system objects.

(3) System functionality is expressed in terms of operations or services associated with each object.

(4) Shared data areas are eliminated. Objects communicate by calling on operations (services) offered by other objects rather than sharing variables. This reduces overall system coupling. There is no possibility of unexpected modification to shared information.

(5) Objects may be distributed and may execute either sequentially or in parallel.
Decisions on parallelism need not to be taken at an early stage of the design process.

Many object-oriented design methods have been proposed [1,2,3,4]. However, some design activities common to all these different propositions include:

(1) Identification of the objects in the system along with their attributes and operations;

(2) The organization of objects into an aggregation hierarchy which shows how objects are 'part of' other objects.

(3) The constructions of dynamic object-use diagram that show which object services are used by other objects (object interaction).

(4) The specification of object interfaces.
It should be noted that the nature of these activities is such that they are intermingled rather than carried out in sequence. [15]

3. Representation and Design of SPERU

Our adventure into the representation and design of the SPERU system starts with the process of object identification.

3.1 Object Identification in SPERU

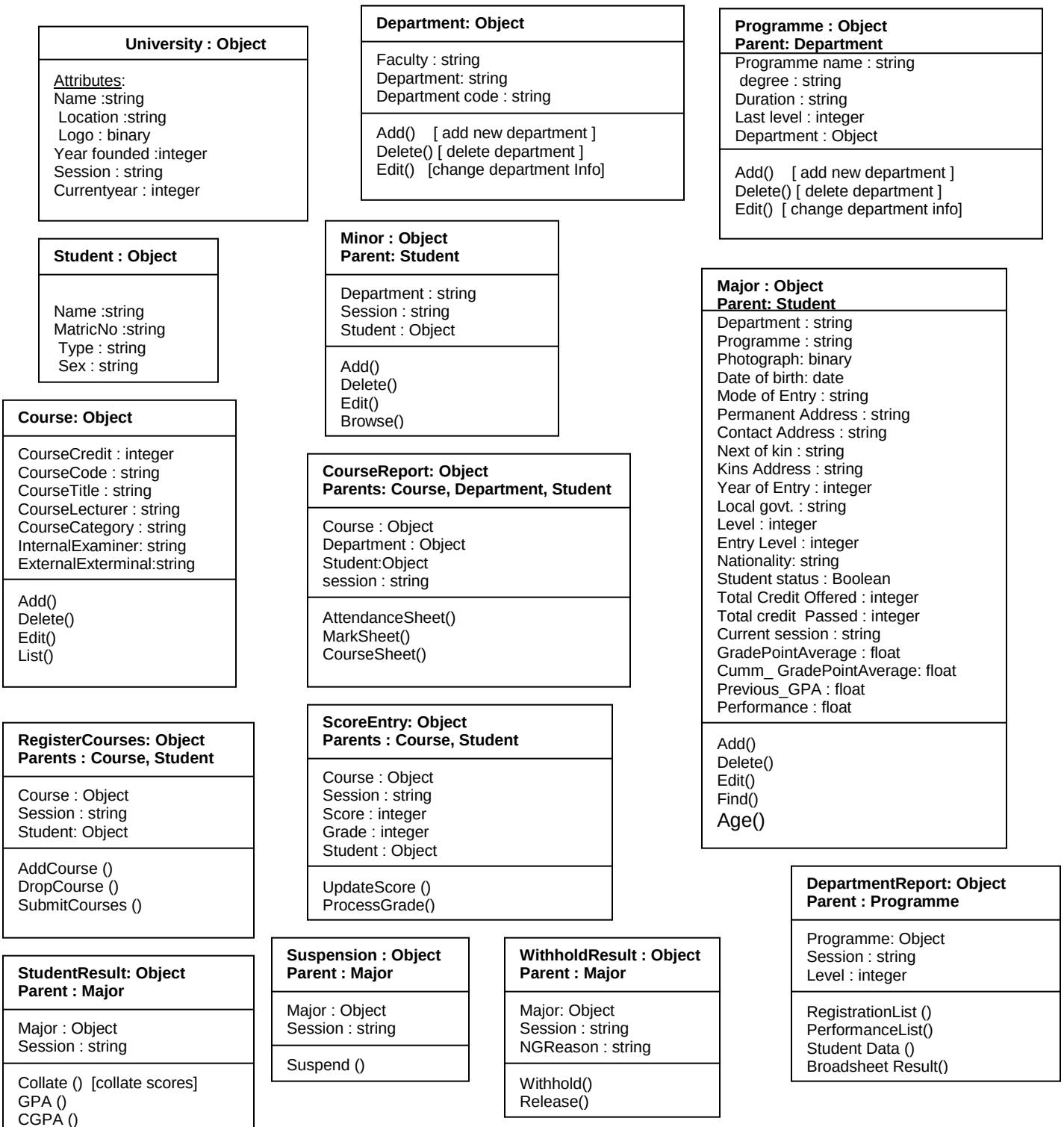
The identification of objects within a system could be achieved using the steps outlined in existing proposals as supported by [1,6,7].

A thorough analysis of the SPERU requirements reveals that the dominant objects within the system are:

University, Department, Student, Course, User, Score Entry, Programme, Course Register, Course

Report and Department report. Some of these objects exist independently, while others are compositions of other objects. Composition is a phenomenon of the object-oriented concept that allows new objects to evolve from existing objects. The object identified are shown in Figure 3.1

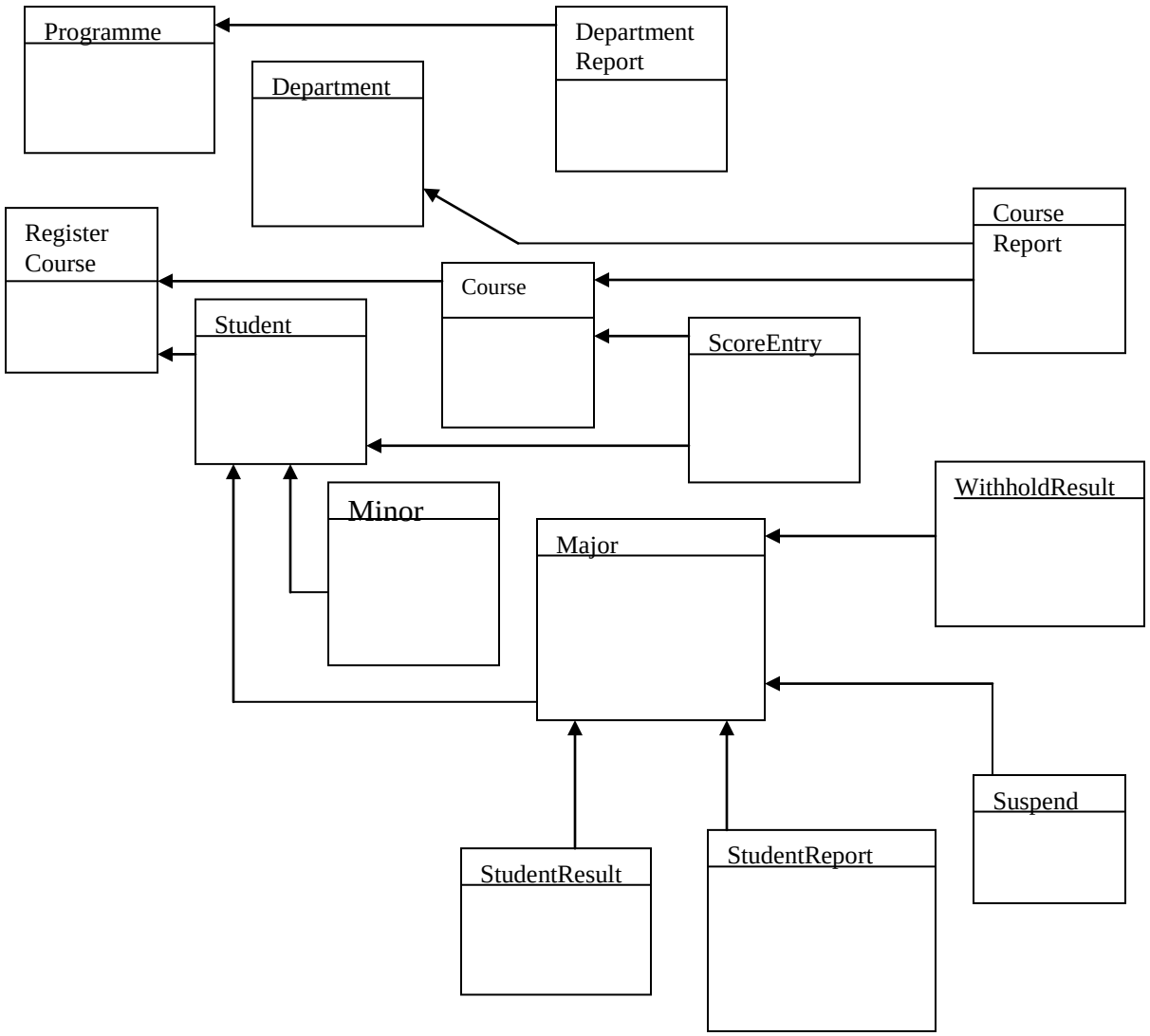
Figure 3.1 **SPERU Objects**



3.2 **Inheritance Hierarchy in SPERU**

A representation of the object class hierarchy showing the inheritance relationship among objects is shown in figure 3.2 after successfully identifying the objects.

Figure 3.2 Class Hierarchy in SPERU



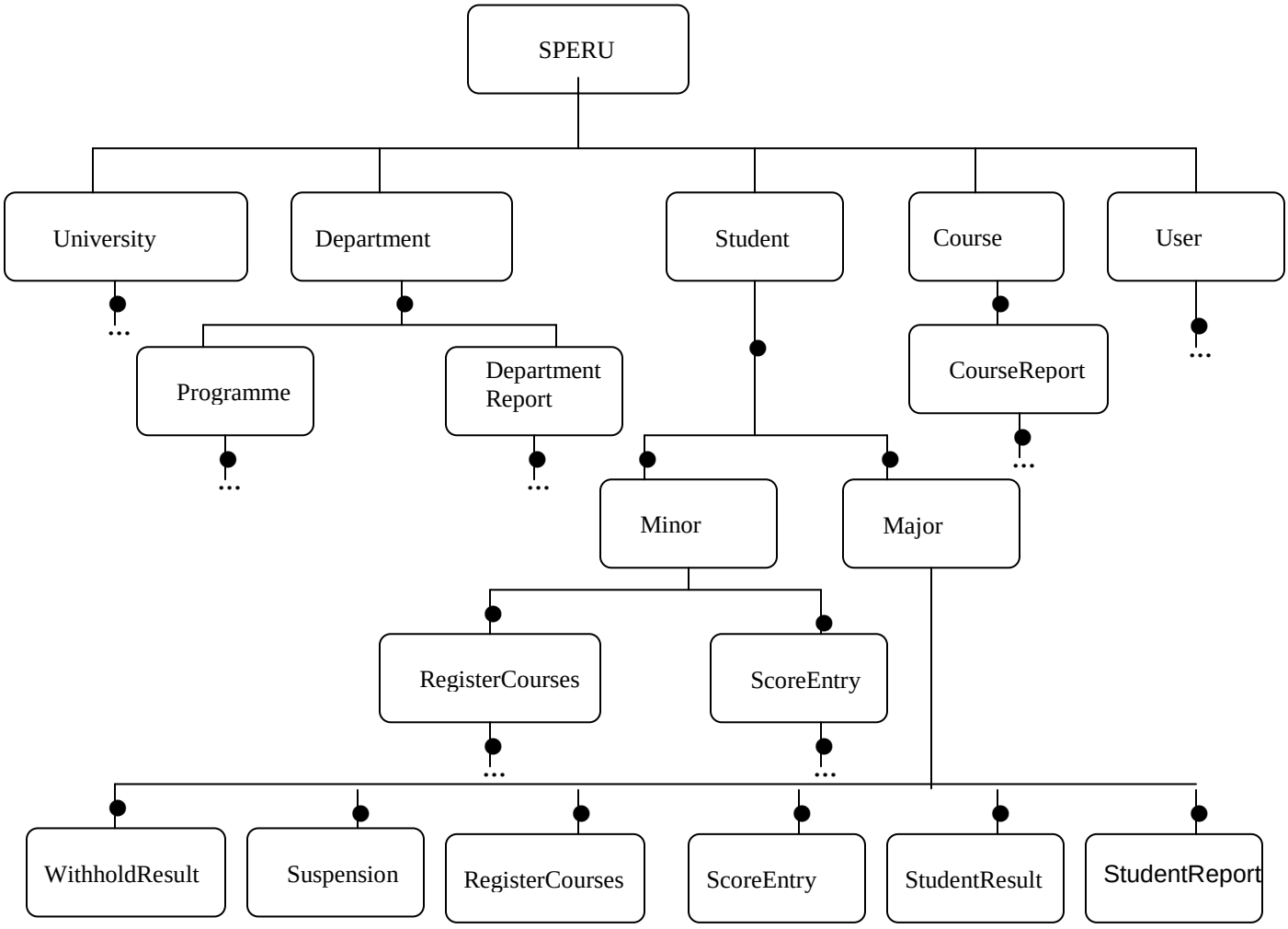
N.B: Rectangular boxes are used to represent objects. The Arrow is indicative of the phrase 'inherits from' .

3.3 **Object Aggregation in SPERU**

Object aggregation is generally used to illustrate the static structure of an object-oriented system. It shows the details of how different objects are 'part of' other objects. This makes it possible to identify objects that can be represented as sub-objects of other objects.

The object aggregation in SPERU is shown in Figure 3.3

Figure 3.3 **Aggregation of SPERU Objects**

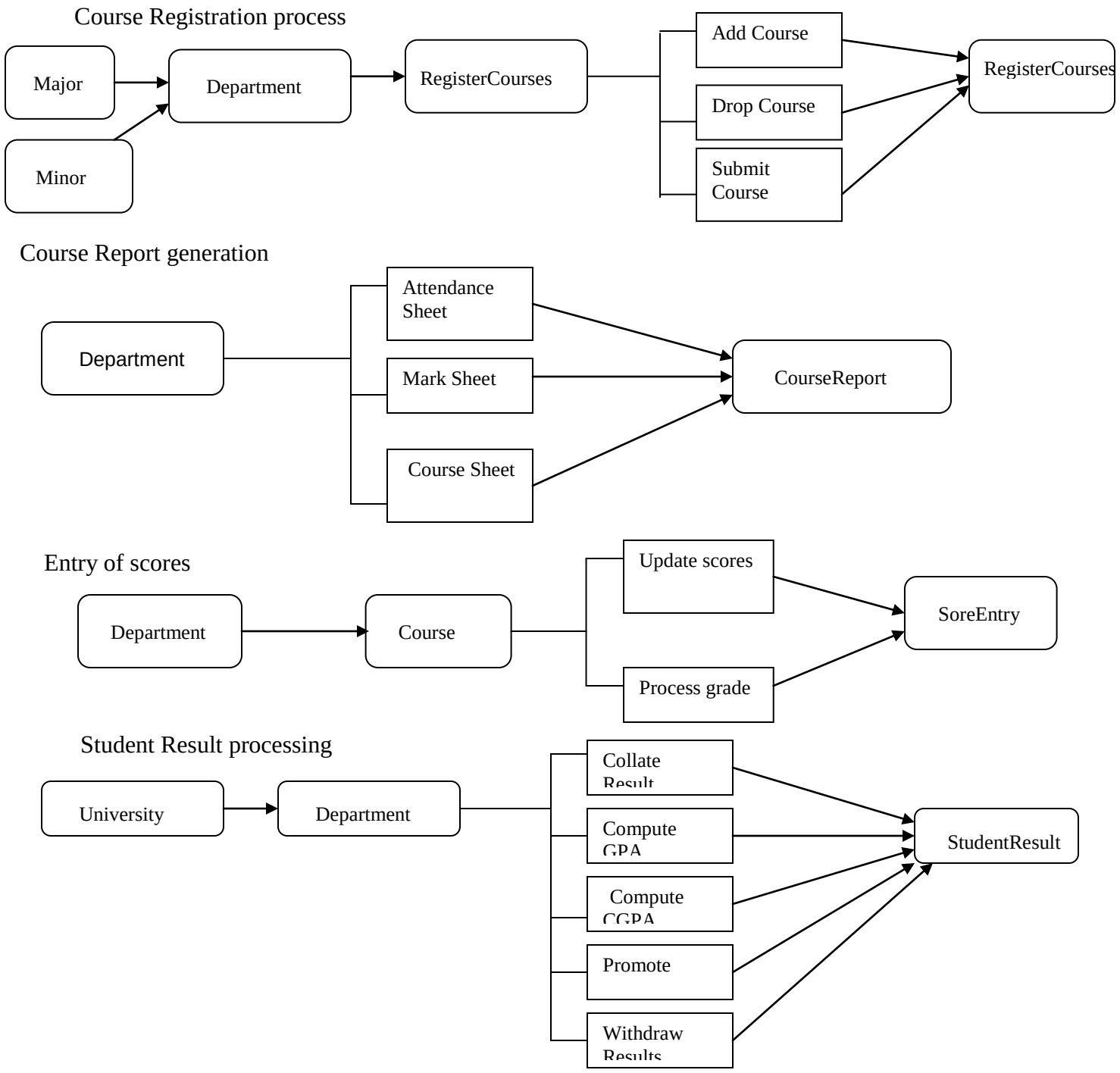


N.B: The aggregation is shown using links annotated with circular blob meaning 'part of'. The ellipses suggest that further details about objects are not shown.

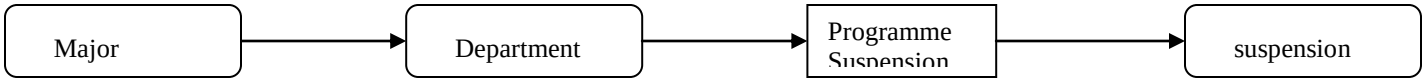
3.4 Object Interaction in SPERU

The Object interaction phase of the design process reveals the dynamic structure of object to object interaction within the system when the system is executing. In order words, it shows how objects interacts with one another, how service calls and requests are passed between different objects. Figure 3.4 shows the object interaction among some of the objects in SPERU.

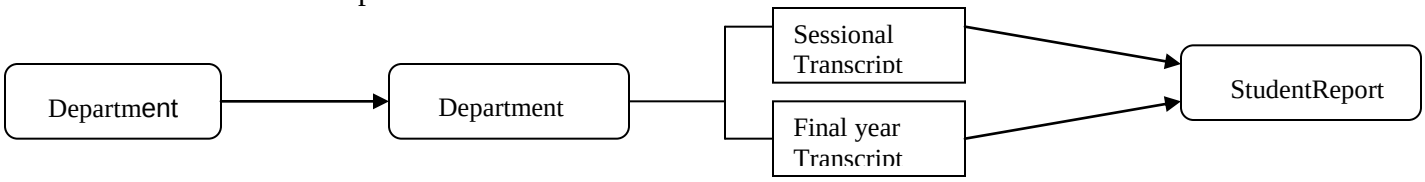
Figure 3.4 Interactions Of SPERU Objects



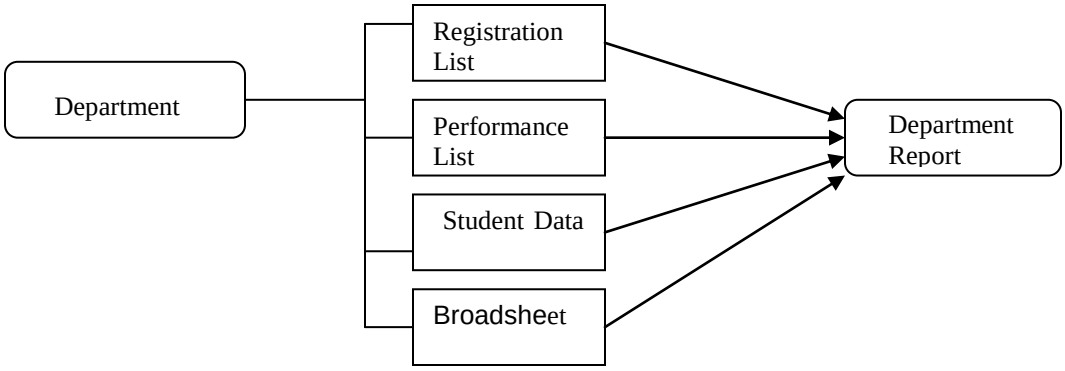
Programme suspension



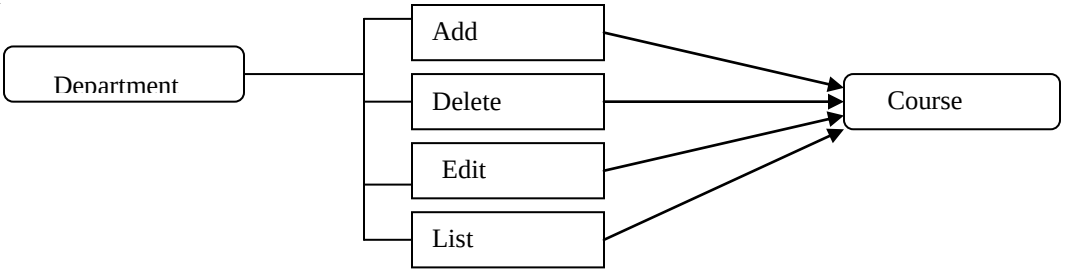
Generation of Student Report



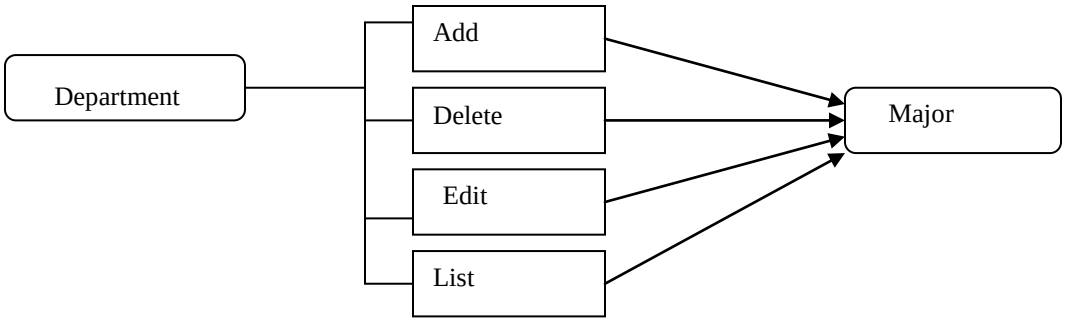
Departmental Reports



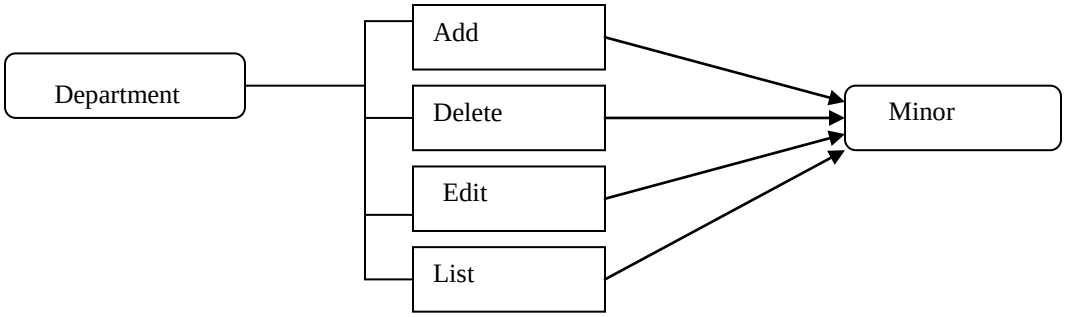
Course Entry process



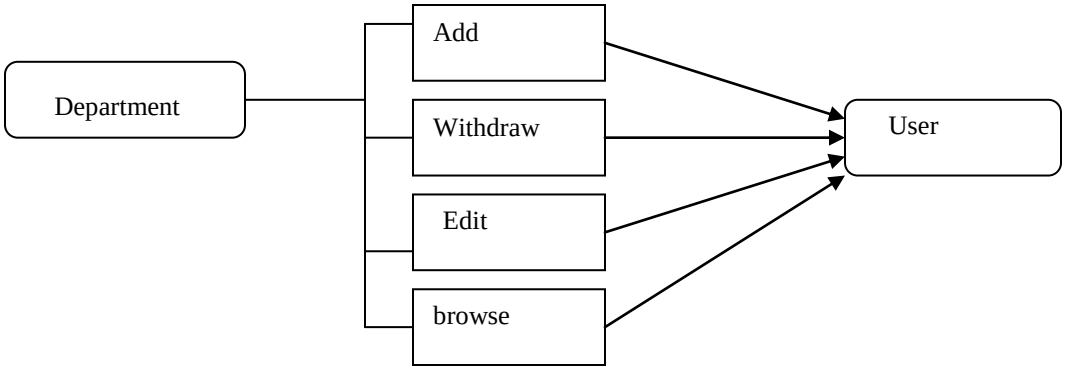
Biodata input for Major students



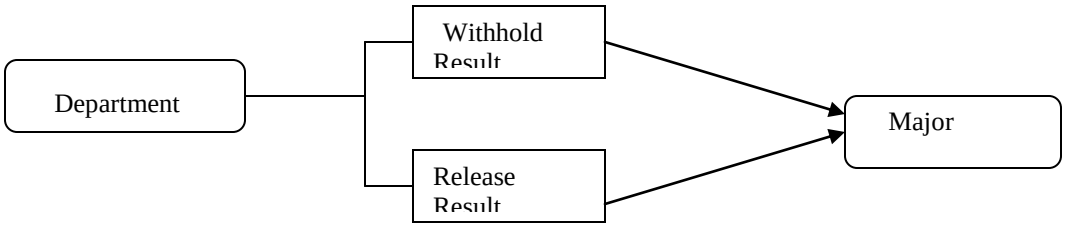
Biodata input for Minor students



Allocation of User access



Withholding of results



N.B: The rectangular boxes give the name of the requested service or operation. The direct arrows indicates direct request from object to object. The arrow shows the direction of service call.

3.5 Object Interface Design for SPERU

Now, that we have represented, in details, the static and dynamic structure of objects within the SPERU , we can now show the specification of the object design interfaces. This involves defining the types of the object attributes, the signatures and semantics of the objection operations as shown below.

Notation : The C++ programming language notation is used to denote the object *interfaces:*

Figure 3.5 : **Object Interfaces in SPERU**

University

```
Class University_Object {
Public:
  char* name;
  char* location;
  char* yearfounded;
  char* session;
  int currentyear;
};
```

Department

```
Class Department_Object {
  Public:
            char* faculty;
            char* Departmentname;
            char* Deptcode;
Private:
            Void Add (char* deptcode);
            Void Delete (char* deptcode);
            Void Edit (char* deptcode);
            };
```

Course

```
Class  Course_Object {
Public
            char * course_code;
            char* course_title;
            char* course_credit
            char*course_category;
            char* course_lecturer;
            char* InternalExaminer;
            char* ExternalExterminer;
            char* course_content;
            char* course_semester;
Private:
             void Add ( char* course_code);
            void Delete ( char* course_code);
            void Edit ( char* course_code);
            void List( char* course_code);
};
```

Programme

```
Class Programme_Object {
Public :
   char* programmename;
  char* degree;
  int duration;
   int lastlevel;
   department_Object Department;
};
```

Minor

```
Class Minor_Object {
Public :
     Department_Object Department;
     Student_Object student;
Private:
    Add ( char* student.matricno);
    Delete ( char* student.matricno);
    Edit ( char* student.matricno);
    Browse ();
};
```

RegisterCourses

```
Class RegisterCourses {
Public:

     Student_Object  ThisStudent;
     char * session;
      Course_object  ThisCourse;
Private:
    void Addcourse (char* ThisCourse.course_code);
    void Dropcourse(char * ThisCourse.course_code);
    void Submitcourses(char* ThisStudent.MatricNo);
};
```

Student

```
Class Student_Object {
Public :
char* name;
char* matricno;
};
```

ScoreEntry

```
Class ScoreEntry {
Public:
Student_Object  ThisStudent;
  char* session;
  Course_object  ThisCourse;
Private:
    void UpdateScore (char* ThisCourse.course_code);
    void ProcessGrade(char *        ThisCourse.course_code);
};
```

Suspension

```
Class Suspend_Object {
Public:
  Major_Object  Majorstudent;
 char* session;
Private:
   Suspend(Major_Object Majorstudent , char* session);
  };
```

Major

```
Class Major_object {
      Public:
            Student_Object student;
            Programme_Object programme;
      Private:
            Char* fullname;
             Unsigned photograph;
            char * date_of_Birth;
            char* mode_of_entry;
            char* perm_address;
            char* cont_address;
            char* Next_of_kin
            char* kin_address;
            char* local_govt;
            char* nationality;
            char* currentsession;
            Bool  student_status;
            int TotCreditOfferd;
            int TotCreditPassed;
            float GPA;
            float CGPA;
            float PGPA;
            char * Perfomance;

             void Add( char * student.matricno);
            void delete( char *
      student.matricno);
            void Edit( char * student.matricno);
             void Age( char * student.matricno);
};
```

Department

```
Class User {
  Public:
            char* Userid;
            char* Accesscode;
            char* Assesslevel;
    Private:
            Void Add (char* Accesscode);
            Void Delete (char* Accesscode);
            Void Edit (char* Accesscode);
};
```

CourseReport

```
Class CourseReport {
Public:

 Course_Object  ThisCourse;
  Student_Object ThisStudent;
 Department_Object HostDepartment;
char* session;
Private:
     void AttendanceSheet (Department_Object HostDepartment, Course_Object
      ThisCourse,  Student_Object ThisStudent, char* session);
    float MarkSheet(Department_Object HostDepartment ,Course_Object  ThisCourse,
    Student_Object ThisStudent, char* session);
    float CourseSheetResult(Department_Object HostDepartment ,Course_Object
    ThisCourse,  Student_Object ThisStudent, char* session);
};
```

StudentResult

```
Class StudentResult {
Public:
  Major_Object  Majorstudent;
 char* session;
Private:
    void Collate (Major_Object Majorstudent , char* session);
   float GPA (Major_Object Majorstudent , char* session);
   float CGPA (Major_Object Majorstudent , char* session);
};
```

DepartmentReport

```
Class DepartmentReport {
Public:
  Programme_Object  ThisProgramme;
 char* session;
Private:
    void RegistrationList (Programme_Object  ThisProgramme , char* session);
   float PerformanceList(Programme_Object  ThisProgramme , char* session);
   float BroadsheetResult(Programme_Object  ThisProgramme , char* session);
 };
```

StudentReport

```
Class StudentReport {
Public:
  Major_Object  Majorstudent;
 char* session;
Private:
    void sessionalTranscript (Major_Object Majorstudent , char*
session);
   float FinalYearResult(Major_Object Majorstudent , char* session);
 };
```

After the specification of the Object interfaces, SPERU was implemented using a rapid application development tool (programming language) known as Borland C++ Builder.

Borland C++ Builder is an object-oriented programming language that contain in totality all the features of C++ spiced with many visual and non–visual programming tools to make an ideal RAD programming environment. It provides a platform for the development of applications that combine the awesome power of the C++ language together with the flair and fun of the Windows environment.

4. QUALITY ASSESSMENT PARAMETERS APPLIED TO SPERU

The main objective of the software design process is to produce a good quality designs that are cost effective to implement and maintain. In this section we apply relevant software metrics to the SPERU system to predict its product quality.

4.1 Design Quality Metrics Applied to SPERU

A software metric is any type of measurement, which relates to a software system. Software engineering metrics are used to characterize software engineering product e.g. design, program codes, test cases, process etc. to determine their success or failure. Metrics could be predictive (predictor metrics) or control metrics. Predictor metric predicts *product quality* while control metrics provide information about *process quality*. [7] The key emphases of the design quality assessment are correctness and maintainability. However, maintainability cannot be measured directly, it is rather closely linked to the following four main attributes: cohesion, coupling, understandability and adaptability which when investigated in respect of SPERU, reveals as follows:

4.1.1 Cohesion in SPERU

The class hierarchy diagram in Figure 3.2, reveals a high degree of cohesion. This is a natural feature of most object-oriented systems. The SPERU is composed of individual cohesive objects each of which encapsulates its own attributes and operations. The objects: *Course, Department University*, and *Student* have the highest level of cohesion within SPERU because they are all super classes. While Objects: *CourseReport, StudentReport, RegisterCourses, Major, Minor* have a reduced level of cohesion because they inherit attributes from a super class. (Yourdon and Constantine, 1979), noted that the higher the level of inheritance within a system, the lower the level of cohesion. SPERU is therefore a very cohesive system.

4.1.2 Coupling in SPERU

Coupling is closely related to cohesion, infact it is concerned with how independent the components are. It indicates strength of interconnection between components in a design (See Figure 3.2 "Class Hierarchy"). SPERU as an object-oriented system is a loosely coupled system. The nature of objects: *Course, Student, and Department* whose representations are concealed within the object and not made visible to external components makes this true. The inheritance features in objects : CourseReport, *StudentReport, Minor, and Major* produce a different form of coupling, which makes this

objects coupled to their superclasses. Changes made to a superclass are automatically propagated to all subclasses.

4.1.3 Understandability of SPERU

The understandability of a design depends largely on cohesion, coupling, meaningful names, documentation and complexity. Understandability deals with how easy is it to comprehend the design. Complexity deals with how complex is it to implement the components. High complexity implies many relationship between different part of a design component. As illustrated in the SPERU class hierarchy, not many nested object relationships exist, thus, limiting the complexity. The inheritance features of subclasses (*Major, Minor, CourseReport*) concealed some design details which is good for understandability.

4.1.4 Adaptability of SPERU

This is a general estimate of how easy it is to change the design. A loosely coupled system design like SPERU posses high adaptability. New components (objects) can be created which inherit the attributes and operations of original components, and only the attributes and operations, which need to be changed, will be modified.

4.2 Program Quality Metrics Applied to SPERU

Generally, program quality assessments are similar to those of the design. Programs should be free of defects and maintainable. The key predictor metrics we can apply to SPERU are listed below:

4.2.1 Length of Code

This is a measure of the size of the program. Generally, the larger the size of the codes of a program component, the more complex and error prone that component is likely to be. The table below (Table 5.2.1) gives a listing of the length of codes for some object interfaces implemented in C++ Builder environment.

Table 4.2.1 Length of SPERU Program Codes

S/No	Object Implementation	Length of Code (No. of lines)
1.	DepartmentReport	430
2.	ScoreEntry	375
3.	StudentReport	366
4.	StudentResult	344
5.	RegisterCourses	290
6.	CourseReport	206
7.	Major (Biodata)	139
8.	Course	69
9.	User	57
10.	Minor (Biodata)	49
11.	Suspend	32
12.	Programme	30
13.	University	22
14.	Department	22
	Total	**2431**
	Average	**162.07**

SPERU Code size = Average code size * Number of function points.

The application of RAD tools in SPERU has led to a compact code size, which readily projects it, as one that is less error prone.

4.2.2 Cyclomatic Complexity

This is a measure of the control complexity of a program. The number of independent paths in a program can be determined by computing the cyclomatic complexity of the program flow graph (Mccabe, 1976, 1983).

The cyclomatic complexity (CC) of any program graph G can be determined using the formula:

CC(G) = Number (edges) – Number(nodes) + 1

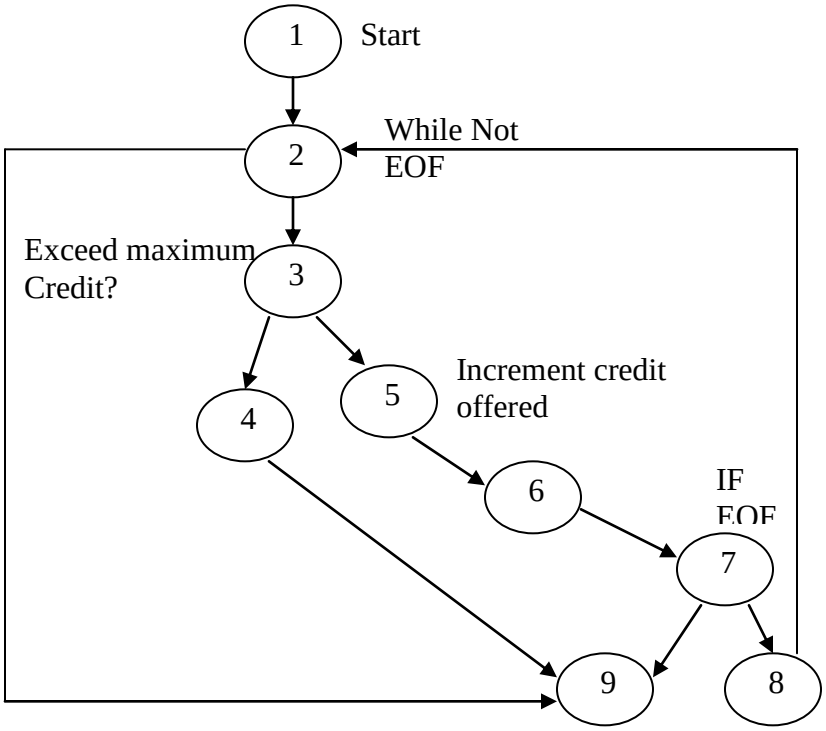
For programs without GOTO statements, the cyclomatic complexity is simply equal to the number of conditions in the program. Compound conditions with N simple predicates are counted as N conditions e.g. if A==B and A==C.

Metric Rule : *The value of the cyclomatic complexity is a measure of the internal complexity of a program. Low value of this metric may correlate with understandable program and design, which also indicates easily maintainable programs.*

Fig 4.2.2 shows the cyclomatic identity of the RegisterCourses interface of SPERU.

Binary Graph for Registercourses Object Interface

Figure 4.2.2



In Figure 4.2.2 ,the independent paths through the binary flow graph are:

(i) 1 2 3 4 9

(ii) 1 2 3 5 6 7 8 2 3 4 9

(iii) 1 2 3 5 6 7 9

If the paths are executed then,

a) we can be sure that every statement in the routine has been executed ones, and

b) every condition has been exercised for true and false condition.

5. Desirable Features of SPERU

The choice of object-oriented software engineering approach for the design of SPERU ,coupled with the application of RAD tools in its implementation brings about a number of attendant benefits. Some of these are:

1. Maintainability

This software is very maintainable. The presence of cohesive objects which are loosely coupled together i.e. they are acceptably independent of each other makes it very easy to

maintain the program. New features could be added to the system by simply modeling them as new objects. Hence, changes to parts of the system could be effected without necessarily starting from the scratch. Also other parts of the system, which need not be affected by such changes are left intact. The understandability and adaptability provided by the nature of the design also promote maintainability.

2. **Reliability**

SPERU has every attribute of a reliable system. The object-oriented model approach to the design and implementation ensures a good level of *fault-tolerance*. Facilities are provided to ensure that operations are allowed to continue when faults cause system failure. Also *fault tracking and detection* is easier because of the object-oriented design of the SPERU system. *Fault avoidance* is also at an appreciable level because the SPERU system was carefully designed and sufficiently tested with the aim of producing a fault-free system.

3. **Usability**

The SPERU software was designed to make it easy to use. It has a built-in help facilities and very friendly user interfaces. The software is window based and event driven, with provision of relevant menu items from which the user can make selection. The SPERU main user interface has such simple modules like *Register, Collate, Produce, Manage* which are simple terms to describe the key activities within the system.

4. **Reusability and Extendibility**

The object-oriented implementation promotes reusability, whenever there is need to extend the SPERU specification to accommodate new user requirements. Also, the existing objects within the system could be readily used to compose some of the new objects desired.

5. **Correctness**

A major plus of the SPERU program is that it fits its requirement. All functionalities specified in the user requirement have been meticulously attended to and provided for in the SPERU software.

6. **Portability**

The SPERU program is quite portable, the optimal code size and optimal usage of system resources like Memory and Hard disk, makes it easy to implement in different environments. The Installable version could be provided on a CD-ROM, which can be distributed to many environments.

7. **Ease in programming Effort**

A lot of ease and relief is brought to the software implementation of the design without compromising speed, efficiency and creativity through the availability of visual RAD tools. These tools helped in conserving a lot of programming time and effort in such activities like multiple interface design, Database access and connection, production of reports and display of records.

8. **Security**

SPERU provides maximum security of data, ensuring that the integrity of data is maintained and restricting unauthorized access.

6.0 Conclusion

The example of SPERU as a sample case of application of object-oriented software engineering principles provides an instance of a new and more efficient concept in the creation of software products. It also attests to the beauty and brilliance of the object-oriented paradigm in the modelling of real-world problems like the Undergraduate Registration and Examination Processing in Tertiary institutions.

References:

1. Coad, P., and Yourdon, E., (1991): Object-oriented analysis, 2nd ed., Yourdon Press, Prentice-Hall, Englewood Cliffs, N.J. pp [86-256]
2. Robinson, P.J. (1992) : Hierarchical object-oriented design; Prentice-Hall, Englewood Cliffs, N.J. pp [213-249]
3. Jacobsen, I., Christensen M.; Johnson, P., and Overgaard, G. (1992): Object-oriented software engineering; Addison- Wesley, England. pp [215-256]
4. Booch, G. (1994): Object-oriented analysis and design; Benjamin-Cummings, U.S.A pp [107-215]
5. Somerville, I. (1998): Software engineering; Addison Wesley; England. pp [207-285]
6. Abbort, R. (1983): Program design by informal English descriptions; Communications of the ACM; 26(11), 882-94 [256]
7. Shlaer, S., and Mellor, S.J. (1998): Object-oriented systems analysis; Yourdon press, Englewood Cliffs, N.J. pp [256]

8. Fenton, N.E.(1991): Software metrics; A rigorous approach; Chapman and Hall, England. pp [132]
9. Yourdon E. and Constantine, L.L.(1979) : Structured design; Prentice-Hall, Englewood Cliffs N.J. pp [213-630]
10. McCabe, T.J. (1976): A complexity measure, IEEE transaction on software engineering; SE–2 pp [308-320]
11. McCabe, T.J. (1983): A Cyclomatic complexity measure, IEEE transaction on software engineering; Vol. 9.
12. Budgen, D (1994): Software design; Addison-Wesley, England. pp [200-213]
13. Jamsa, K. and Klander, L. (1998): C/C++ Programmers Bible; Jamsa Press, Las Vegas, U.S.A.
14. Ince, D. (1994): ISO9001 and software quality assurance; MCGraw-Hill, England. pp [613,676]
15. Davies, A.M. (1993): Software requirements: Objects, Functions and States; Prentice-Hall; U.S.A, pp [64]