# University Exam Scheduling

Prepared by

|  |  |  |
| --- | --- | --- |
| Vedant Sahai | 8364 | vedantsahai18@gmail.com |
| Pratik Chowdhury | 8322 | pratikc@live.co.uk |
| Elvis Dsouza | 8333 | elvisejsondsouza57@gmail.com |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |
| --- | --- |
| Instructor: | Dr. B S Daga |
| Course: | Software Engineering |
|  |  |
|  |  |
| Date: | 7th April 2020 |

**Index**

|  |  |  |
| --- | --- | --- |
| **#** | **Experiment Name** | **Aim** |
| 1 | SOFTWARE REQUIREMENTS SPECIFICATION | To prepare software requirements specification document for a selected case study in IEEE format |
| 2 | FUNCTION POINT CALCULATION | To calculate function point for a selected case study |
| 3 | COST ESTIMATION USING COCOMO MODEL | To calculate function point for a selected case study |
| 4 | PROJECT SCHEDULING USING PROJECT MANAGEMENT TOOL | Use project management tool to schedule project plan for a selected case study |
| 5 | RISK MANAGEMENT MONITORING AND MITIGATION PLAN | Develop a risk table for a selected case study |
| 6 | General Test-Driven Development | General test driven development for a selected case study |
| 7 | BLACK BOX TESTING | To design test cases for performing black box testing (equivalence partitioning and boundary value analysis) for a selected case study |
| 8 | WHITE BOX TESTING | To design test cases for performing white box testing for a selected case study |
| 9 | COHESION AND COUPLING | Draw Architecture diagram and incorporate Cohesion and Coupling for each module of <type your selected case study |
| 10 | DESIGN PATTERNS | Application of at least two types of design patterns in selected case study |

|  |  |
| --- | --- |
| **Practical No** | 01 |
| **Title** | Software requirement Specification |
| **Date Of Performance** |  |
| **Date Of Submission** |  |
| **Roll No** | 8322 |
| **Name Of The Student** | CHOWDHURY PRATIK VINAYAK |

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| 1 | On Time Completion(2M) |  |
| 2 | Completeness(4M) |  |
| 3 | Correctness(4M) |  |
| 5 | Total (10M) |  |

**EXPERIMENT NO. 1**

**SOFTWARE REQUIREMENTS SPECIFICATION**

**Aim**

To prepare software requirements specification document for a University Exam Scheduling in IEEE format.

**Description**

An SRS is basically an organization's understanding (in writing) of a customer or potential client's system requirements and dependencies.

An SRS should address the following:

a) **Functionality**. What is the software supposed to do?

b) **External interfaces.** How does the software interact with people, the system’s hardware, other hardware, and other software?

c) **Performance.** What is the speed, availability, response time, recovery time of various software functions, etc.?

d) **Attributes.** What are the portability, correctness, maintainability, security, etc. considerations?

e) **Design constraints** imposed on an implementation. Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.

**Conclusion**

Based on the information we had, we prepared an SRS which helped us to simplify our tasks, ensure the goals were known before hand and helped ensure that all involved parties were aware about what tasks were to be handled and how

# Software Requirements Specification for University Exam Scheduling

Prepared by

|  |  |  |
| --- | --- | --- |
| Vedant Sahai | 8364 | vedantsahai18@gmail.com |
| Pratik Chowdhury | 8322 | pratikc@live.co.uk |
| Elvis Dsouza | 8333 | elvisejsondsouza57@gmail.com |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |
| --- | --- |
| Instructor: | Dr. B S Daga |
| Course: | Software Engineering |
|  |  |
|  |  |
| Date: | 7th April 2020 |

**Table of Contents**

|  |  |  |
| --- | --- | --- |
|  | **Abstract** | 3 |
| **1.** | **Introduction** | |
| 1.1 | Purpose | 4 |
| 1.2 | Scope | 4 |
| 1.3 | Definition, Acronyms, Abbreviations | 4 |
| 1.4 | References | 5 |
| 1.5 | Developer's Responsibilities | 5 |
| **2.** | **General Description** | |
| 2.1 | Product Function Overview | 6 |
| 2.2 | User Characteristics | 6 |
| 2.3 | General Constraints | 6 |
| 2.4 | General Assumptions & Dependencies | 7 |
| **3.** | **Specific Requirements** | |
| 3.1 | Inputs & Outputs | 8 |
| 3.2 | Functional Requirements | 9 |
| 3.3 | External Interface Requirements | 9 |
| 3.4 | Performance Constraints | 9 |
| 3.5 | Design Constraints | 10 |
|  | 3.5.1    Software Constraints | 10 |
|  | 3.5.2    Hardware Constraints | 10 |
|  | 3.5.3    Acceptance Criteria | 10 |

**ABSTRACT**

A graph-coloring-based algorithm for the exam scheduling application, with the objective of achieving fairness, accuracy, and optimal exam time period. Through the work, we consider few assumptions and constraints, closely related to the general exam scheduling problem, and mainly driven from accumulated experience at various universities. The performance of the algorithm is also a major concern of this SRS. We are scheduling exams at a university using Graph Coloring. The algorithm can be used for any university, but for testing purposes, we have tuned the program for IIT Kanpur, with data of even semester for the academic year 2014-15.

# INTRODUCTION

## Purpose

The exam schedule is a challenging task that universities and colleges face several times every year. The challenge is to schedule so many exams of courses in a limited, and usually short, the period of time. An Exam schedule should avoid conflicts, in the sense that no two or more exams for the same student are scheduled at the same time. Part of the challenge is to achieve fairness for the students. A fair schedule does not schedule more than two exams, for example for a student on one day. In the meantime, a fair schedule does not leave a big gap between exams for the students. The exam scheduling problem is defined as follows: "We first represent the courses by nodes of a graph, where 2 nodes are adjacent if the 2 corresponding courses are registered by at least one student. Then, it is required to assign each course represented by a node a time slot, such that no two adjacent nodes have the same slot, in the condition that a set of constraints imposed on the problem are also met." We solve this problem by using the node graph colouring technique. This study provides a mechanism for automatic exam-schedule generation that achieves fairness and minimizes the exam period. As a result, this paper presents a graph-colouring-based algorithm for the exam scheduling application which achieves the objectives of fairness, accuracy, and optimal exam time period.

## Scope

This document is the only one that describes the requirements of the system. It is meant for use by the developers and will be the basis for validating the final delivered system. Any changes made to the requirements in the future will have to go through a formal change approval process. The developer is responsible for asking for clarifications, where necessary, and will not make any alterations without the permission of the client. We are scheduling exams at a university using Graph Coloring. The algorithm can be used for any university, but for testing purposes, we have tuned the program for IIT Kanpur, with data of even semester for the academic year 2014-15.

## Definitions, Acronyms, Abbreviations

Not applicable.

## References

[1] Alon N., “A Note on Graph Colorings and Graph Polynomials,” Journal of Combinatorial Theory Series B, vol. 70, no. 1, pp. 197-201, 1997.

[2] Baldi P., “On a Generalized Family of Colorings,” Graphs and Combinatorics, vol. 6, no. 2, 1990.

[3] Bang-Jensen J. and Gutin G., Digraphs: Theory, Algorithms and Applications, Springer-Verlag, 2000.

[4] Bar-Noy A., Motwani R., and Naor J., “The Greedy Algorithm is Optimal for On-line Edge Coloring,” Information Processing Letters, vol. 44, no. 5, pp. 251-253, 1992.

[5] Batenburg K. and Palenstijn W., “A New Exam Timetabling Algorithm,” Leiden Institute of Advanced Computer Science (LIACS), [http://visielab.ua.ac.be/staff/batenburg/papers/ba pa\_bnaic\_2003.pdf](http://visielab.ua.ac.be/staff/batenburg/papers/ba%20pa_bnaic_2003.pdf).

[6] Bauernoppel F. and Jung H., “Fast Parallel Vertex Coloring,” in L. Budach (eds.), Fundamentals of Computation Theory, FCT '85, Cottbus, GDR, Sept, 1985, vol. 199 of Lecture Notes in Computer Science, pp. 28-35, SpringerVerlag, Berlin, 1985.

[7] Bean D., “Effective Coloration,” The Journal of Symbolic Logic, vol. 41, no.2, pp. 469-480, June 1976.

[8] Burke E. and Petrovic S., “Recent Research Directions in Automated Timetabling,” European Journal of Operational Research (EJOR), vol. 140, no. 2, pp 266-280, 2002

## Developer’s Responsibilities

The developer is responsible for

(a) developing the system,

(b) installing the software on the client’s hardware,

(c) conducting any user training that might be needed for using the system, and (d) maintaining the system for a period of one year after installation.

# GENERAL DESCRIPTION

## Product Functions Overview

In the first-year department, there are a set of classrooms. Every semester the department offers courses, which are chosen from the set of department courses. A course has expected enrollment and could be for graduate students or undergraduate students. For each course, the instructor gives some time preferences for lectures.

The system is to produce a schedule for the department that specifies the time and room assignments for the different courses. Preference should be given to graduate courses, and no two graduate courses should be scheduled at the same time. If some courses cannot be scheduled, the system should produce a “conflict report” that lists the courses that cannot be scheduled and the reasons for the inability to schedule them.

## User Characteristics

The main users of this system will be department secretaries, who are somewhat literate with computers and can use programs such as editors and text processors.

## General Constraints

The system should run on Sun 3/50 workstations running UNIX 4.2 BSD. The scheduling problem has its own peculiarities, which have to be taken into consideration at the implementation level. For example, the node with a large degree represents a course in which many students are registered to many other courses (different group of students may be registered to different courses). Also, nodes with large degrees have a large number of students as well. In order to have an efficient schedule; the nodes with larger degrees should be coloured first. Giving priority to the nodes with the larger degrees is in line with typical university schedules which tend to schedule the university required courses early in the exam period. The nodes representing university and college requirement courses have large degrees. The weight of an edge indicates the number of common students registered at both courses (nodes) connected to that edge. Giving priority in the colouring algorithm to nodes connected to a large weight-edge will enable a solution optimization geared towards the larger groups of students. Another point to consider before we describe the algorithm is the multi-section courses. Multi-sections of a multi-sections course should be scheduled at the same time, and thus the corresponding nodes should have one colour. Also, they typically occupy several halls. The number of halls used by a course has an impact on the concurrency level per time slot. When such multi-sections are scheduled for a time slot, i.e. assigned a colour, the concurrency level is to be reduced by the number of sections for that course. For implementation purposes, we augment the nodes of the graph with a value equal to the number of sections in the course; we shall call this value the course concurrency level CL (ci). Thus, we assign a concurrency limit for each colour Np (CIJ). After assigning a colour to a node Ci, we reduce the concurrency limit of the colour by CL (ci). The concurrency limit is set by the registrar and depends on the number of available halls, and staff to monitor the exams.

## General Assumptions and Dependencies

We summarize the main assumptions and constraints as follows: 1. The number of exam periods per day (Time Slots (TS)) can be set by the user. TS depend on college/department-specific constraints. For example, a university that uses a 2-hours exam period and begins the exam day at 8:00 am and finish at 8:00 pm may set TS to 5. 2. The number of concurrent exam sessions or concurrency level (Np) depends on the number of available halls, and the availability of faculty to conduct the exams. Np is determined by the registrar’s office. This paper assumes that Np is a system parameter and the scheduling algorithm has been examined with several Np values. A New Exam Scheduling Algorithm Using Graph Coloring 81 3. A student shall not have more than (y) exams per day (fairness requirement) and is treated as a system tunable parameter. 4. A student shall not have a gap of more than (x) days between two successive exams, and this factor is to be determined by the college or department (another fairness requirement). 5. The schedule shall be done in the minimal possible period of time, i.e., minimize the number of exam slots and/or the number of exam days. The exam time period is an outcome of the scheduling algorithm. 6. Next, we give some more definitions that are relevant to the underlined problem. Let C be a list of all courses to be scheduled. The length of this list is n. In other words, n is the number of courses on the list. A course at the position I in the list C is referred to using an index ci. Let G be the graph that represents the list C of courses. We impose a weight wij to each edge of G, where wij is defined as the number of students present in both courses ci and cj . An edge eij exists between nodes ci and cj iff wij is not 0. We define a weight matrix W to be an nxn matrix, where n is the number of courses to be scheduled for the exams, and wij equals the weight of the edge eij that joins the courses ci and cj . Such a weight imposed on the edges of G represents the exam conflict complexity present in courses ci and cj. A multi-section course is considered as one course. However, the number of sections per course is taken into consideration in the process of hall assignment.

# SPECIFIC REQUIREMENTS

## Inputs and Outputs

*INPUT –*

(a) Course Data: Course Code, Students Enrolled, Room Numbers

(b) Lecture Hall Data: Seats available in odd and even rows respectively, for exam purposes.

(c) Current Schedule: Exam schedule as allocated by the university, for comparison purposes.

The system has three file inputs and produces three types of outputs.

*Input file 1:* Contains the list of room numbers and their capacity; a list of all the courses in the department catalogue; The format of the file is:

rooms

room1: cap1 room2: cap2

:

;

students

student1:student2: cap2

courses

course1 , course2 , course3 , ;

*Input file 2:* Contains information about the seats offered. For each course, it specifies the seat number available, expected enrollment, and a number of lecture time preferences. A course number greater than 600 is a post-graduate course; the rest are undergraduate courses. The format of this file is:

seat enrollment semester preferences

c#1 cap1 pre1, pre2, pre3 ... c#2 cap2 pre1 , pre2 , pre3 ...

*Input file 3*: Exam schedule as allocated by the university, for comparison purposes.

#### OUTPUT—

(a) Day and Slot allotted for all courses

(b) Lecture Hall and seating arrangement (odd or even) for all courses.

(c) An allotment is done taking care of no clash for any students, and constraints to make the exam schedule convenient for most students.

## Functional Requirements

1. Determine the time and room number for the courses such that the following constraints are satisfied:

* No more than one course should be scheduled at the same time in the same room.
* The classroom capacity should be more than the expected enrollment of the course.
* Preference is given to post-graduate courses over undergraduate courses for scheduling.
* The post-graduate (undergraduate) courses should be scheduled in the order they appear in the input file, and the highest possible priority of an instructor should be given. If no priority is specified, any class and time can be assigned. If any priority is incorrect, it is to be discarded.
* No two post-graduate courses should be scheduled at the same time.
* If no preference is specified for a course, the course should be scheduled in any manner that does not violate these constraints.

1. Produce a list of all courses that could not be scheduled because some constraint(s) could not be satisfied and give reasons for unschedulability.
2. The data in input file 2 should be checked for validity against the data provided in input file 1. Where possible, the validity of the data in input file 1 should also be checked. Messages should be given for improper input data, and the invalid data item should be ignored.

## External Interface Requirements

*User Interface:*Only one user command is required. The file names can be specified in the command line itself or the system should prompt for the input file names.

## Performance Constraints

For input file 2 containing 20 courses and up to 5 preferences for each course, the reports should be printed in less than 1 minute. Our algorithm has linear complexity, except when (ρ = n-1) and hence a polynomial solution of the second degree. We prove the following: Lemma: The algorithm described above achieves a minimal number of colours when the upper bound of colours is given by the clique (largest completely connected sub-graph). Proof: A completely connected graph with size K requires K+1 colour. The algorithm detects the clique in the graph. The algorithm also detects the clique related to each node in the graph starting from the node with the largest degree. Then, the algorithm colours the largest completely connected sub-graphs first, thus utilizing the minimal available colours to colour the subgraphs. For each node, the algorithm will not use more colours than those required by the largest completely connected sub-graph. Thus the largest number of colours used by the algorithm is only that required by the largest sub-graph, which is the absolute minimum possible number of colours.

## Design Constraints

### Software Constraints

The system is to run under the UNIX operating system.

### Hardware Constraints

The system will run on a Sun workstation with 256 MB RAM, running UNIX. It will be connected to an 8-page-per-minute printer.

### Acceptance Criteria

The algorithm Color Schedule was applied to a course list of a university. The number of courses in the testbed is 546 with an average of 2 sections per course, for a total of 1092 exam sessions to schedule. The graph produced for the courses has an average degree of 54 and a maximum degree of 434. The coloring algorithm completed in 90 seconds (almost the same for all runs). We ran the algorithm with different parameters. The variables are the number of exam slots per day (3, 4, 5, 6, 7). The concurrency limit is varied between 10 and 100. The constraint is that a student will not have more than 2 exams per day. The results for the various runs of the algorithm are plotted in Figure 2 below. The registrar office can use the plots to decide on the number of days and number of exam sessions per day for the schedule. For example, with 7 exam slots per day, the exam period can be completed in 12 days with 50 sessions per day. Note that the registrar office can produce several schedules in a short period of time (90 seconds per schedule) and select the appropriate schedule. Figure 3 shows the time analysis performance of the algorithm. Note that the execution time is a linear function of the number of courses. The average degree of the graph is also shown in the figure. The average degree does not increase at the same rate as the number of courses. This is typical of university courses.

|  |  |
| --- | --- |
| **Practical No** | 02 |
| **Title** | Cost Estimation(Function Point Analysis) |
| **Date Of Performance** |  |
| **Date Of Submission** |  |
| **Roll No** | 8322 |
| **Name Of The Student** | CHOWDHURY PRATIK VINAYAK |

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| 1 | On Time Completion(2M) |  |
| 2 | Accurate Time and size estimation (4M) |  |
| 3 | Postlab(4M) |  |
| 5 | Total (10M) |  |

**EXPERIMENT NO. 2**

**FUNCTION POINT CALCULATION**

**Aim**

To calculate function point for University Exam Scheduling

**Description**

In this method, the number and type of functions supported by the software are utilized to find FPC(function point count).

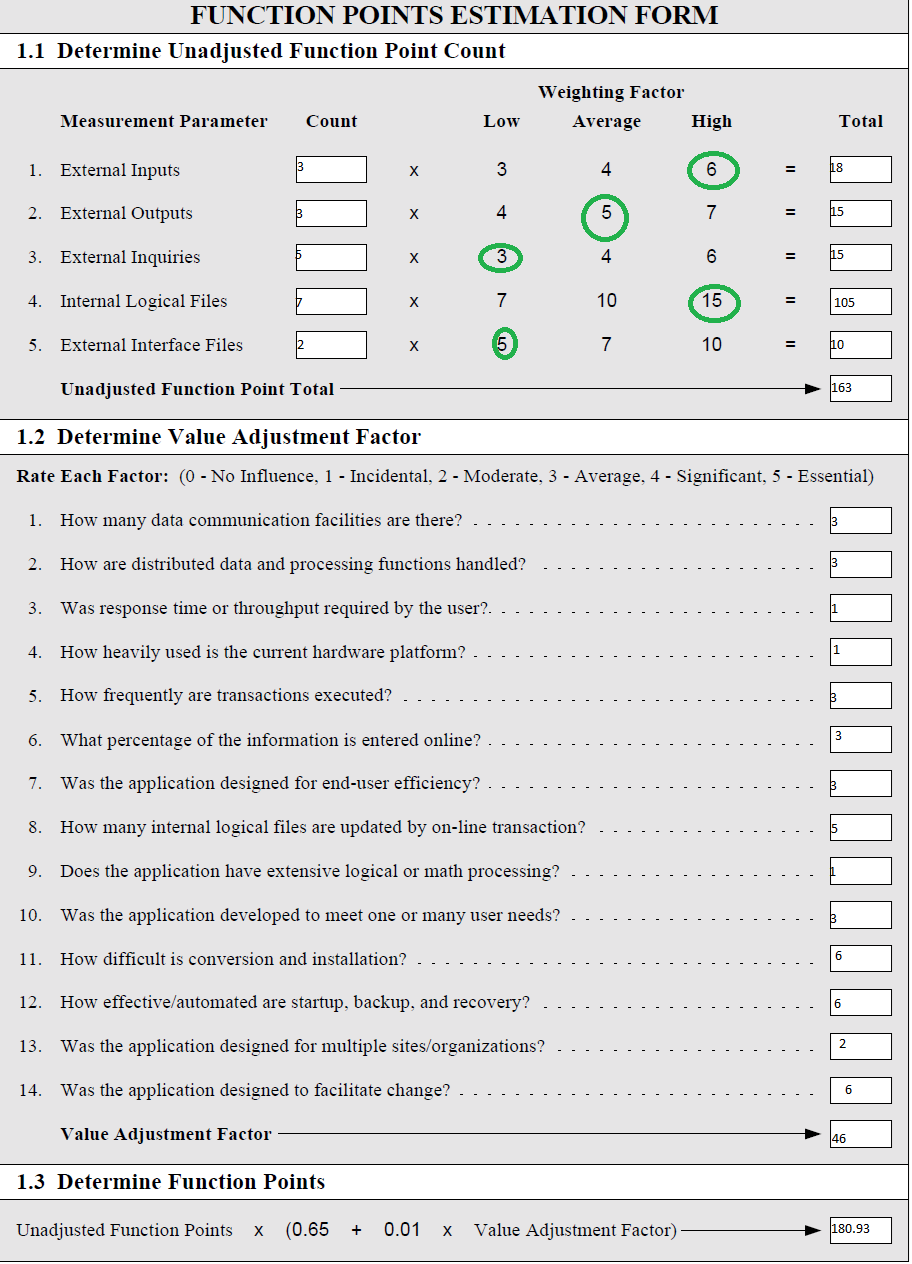
The steps in function point analysis are:

* Count the number of functions of each proposed type.
* Compute the Unadjusted Function Points(UFP).
* Find Total Degree of Influence(TDI).
* Compute Value Adjustment Factor(VAF).
* Find the Function Point Count(FPC).

The explanation of above points given below:

* **Count the number of functions of each proposed type:**
* Find the number of functions belonging to the following types:
  + External Inputs: Functions related to data entering the system.
  + External outputs: Functions related to data exiting the system.
  + External Inquiries: They leads to data retrieval from system but don’t change the system.
  + Internal Files: Logical files maintained within the system. Log files are not included here.
  + External interface Files: These are logical files for other applications which are used by our system.
* **Compute the Unadjusted Function Points (UFP)**: Categorise each of the five function types as simple, average or complex based on their complexity. Multiply count of each function type with its weighting factor and find the weighted sum.
* **Find Total Degree of Influence**: Use the ’14 general characteristics’ of a system to find the degree of influence of each of them. The sum of all 14 degrees of influences will give the TDI. The range of TDI is 0 to 70. The 14 general characteristics are: Data Communications, Distributed Data Processing, Performance, Heavily Used Configuration, Transaction Rate, On-Line Data Entry, End-user Efficiency, Online Update, Complex Processing Reusability, Installation Ease, Operational Ease, Multiple Sites and Facilitate Change. Each of above characteristics is evaluated on a scale of 0-5.
* **Compute Value Adjustment Factor(VAF)**: Use the following formula to calculate VAF VAF = (TDI \* 0.01) + 0.65
* **Find the Function Point Count**: Use the following formula to calculate FPC FPC = UFP \* VAF

**CONCLUSION**

As can be seen via the calculations performed based on the formulas given in the FPE Form, it was found that the FPE for University Exam Scheduling was approximately 180.93

|  |  |
| --- | --- |
| **Practical No** | 03 |
| **Title** | Cost Estimation (COCOMO) |
| **Date Of Performance** |  |
| **Date Of Submission** |  |
| **Roll No** | 8322 |
| **Name Of The Student** | CHOWDHURY PRATIK VINAYAK |

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| 1 | On Time Completion(2M) |  |
| 2 | Accurate Time and size estimation (4M) |  |
| 3 | Postlab(4M) |  |
| 5 | Total (10M) |  |

**EXPERIMENT NO. 3**

**COST ESTIMATION USING COCOMO MODEL**

**Aim**To estimate project cost using COCOMO Model for University Exam Scheduling.

**Description**

**Constructive Cost Model (COCOMO)**

* COCOMO is one of the most widely used software estimation models in the world.
* This model is developed in 1981 by Barry Boehm to give estimation of number of man-months it will take to develop a software product.
* COCOMO predicts the efforts and schedule of software product based on size of software.

Similarly, there are three classes of software projects.

* Organic mode: In this mode, relatively simple, small software projects with a small team are handled. Such team should have good application experience to fewer rigid requirements.
* Semi-detached projects: In this class intermediate project in which team with mixed experience level are handled. Such project may have mix of rigid and less than rigid requirements.
* Embedded projects: In this class, project with tight hardware, software and operational constraints are handled.

Each Model in detail

1. **Basic Model**The basic COCOMO model estimate the software development effort using only Lines of code  
   Various equations in this model are:-
   1. E = a(b) \* KLOC(b,b)  
      KLOC is the estimated number of delivered lines of code for the project  
      Where, E (Effort applied in person-months)
   2. D = c(b) \* E(d,b)  
      D is the development time in chronological months
2. **Intermediate Model**  
   This is extension of COCOMO model.

This estimation model makes use of set of “Cost Driver Attributes” to compute the cost of software.

**I. Product attributes**

1. required software reliability
2. size of application data base
3. complexity of the product

**II. Hardware attributes**

1. run-time performance constraints
2. memory constraints
3. volatility of the virtual machine environment
4. required turnaround time

**III. Personnel attributes**

1. analyst capability
2. software engineer capability
3. applications experience
4. virtual machine experience
5. programming language experience

**IV. Project attributes**

1. use of software tools
2. application of software engineering methods
3. required development schedule

Each of the 15 attributes is rated on a 6-point scale that ranges from "very low" to "extra high" (in importance or value).

The intermediate COCOMO model takes the form  
E = a(i) KLOC(b,i) \* EAF  
Where, E is the effort applied in person-months  
KLOC is the estimated number of delivered lines of code for the project

**3. Detailed COCOMO Model**  
The detailed model uses the same equation for estimation as the intermediate Model.  
But detailed model can estimate the effort (E), duration (D), and person (P) of each of development phases, subsystem and models.

**Conclusion**

For our Project FP = 180.93  
Hence, KLOC = 30 \* 180.93  
 = 5.4k  
Now as we know using Basic COCOMO Model,  
 Effort Applied E = Ab \* KLOCBb = 3 \* (5.4)1.2 = 22.7 Person Months  
 Development Time D = CbEDb = 2.5 \* (22.7)0.32 = 6.8Chronological Months  
 Number of People B = 22.7 / 6.8 = 3 (Approx)

As can be noticed from our requirements, using a B approximately require a team of 3 people

|  |  |
| --- | --- |
| **Practical No** | 04 |
| **Title** | Project Scheduling |
| **Date Of Performance** |  |
| **Date Of Submission** |  |
| **Roll No** | 8322 |
| **Name Of The Student** | CHOWDHURY PRATIK VINAYAK |

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| 1 | Risk Identification(3M) |  |
| 2 | Risk Documentation(2M) |  |
| 3 | Selection of appropriate Mitigation Approach(3M) |  |
| 4 | Thorough Contingency Plan(2M) |  |
| 5 | Total (10M) |  |

**Aim**

Use project management tool to schedule project plan for University Exam Scheduling

**Description**

1. **Project Summary**
   * 1. **Project Overview**The course scheduling software is meant to create a schedule for university for the first year department, given the preferences of professors and the information on available rooms and timeslots for courses.
     2. **Project Scope**The project scope is primarily to create a schedule and give suitable messages from the given preferences and data given in input files. Getting the data to prepare the input files is out of scope of this system.
     3. **Development Process**We follow the waterfall model of software development as it is simple and small.
     4. **Effort, Schedule and Team:**The team comprises of the following 3 persons:
        + *Total Effort: 2.4 person-months (53 person-days)*
        + *Project duration: 3.5 months*
     5. **Assumptions made:**  
        No major assumptions beyond what is stated in the SRS.
2. **Detailed Effort and Schedule**The phase wise estimates were obtained earlier and given in the book. To summarize the total effort is 53 person-days. Of this the distribution is design: 0.4 (9 days), detailed design: 0.6 (13 days), coding: 1.0 (22 days), and integration: 0.4 (9 days).  
   As the project staff (students) are spending on the project about 1/4th to 1/3 rd of their total time, the durations of the tasks have to be suitably fixed. The overall schedule for the project is given below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **#** | **Task** | **Estimated Effort (person -**  **days)** | **Start Date (dd/mm/ yyyy)** | **End date (dd/mm/ yyyy)** | **Person** | **Actual Effort (man-**  **hrs)** |
| 1 | System design | 9 | Jan 18 | Feb 1 | A, B |  |
| 2 | Detailed design | 13 | Feb 1 | Feb 28 | A, B, C |  |

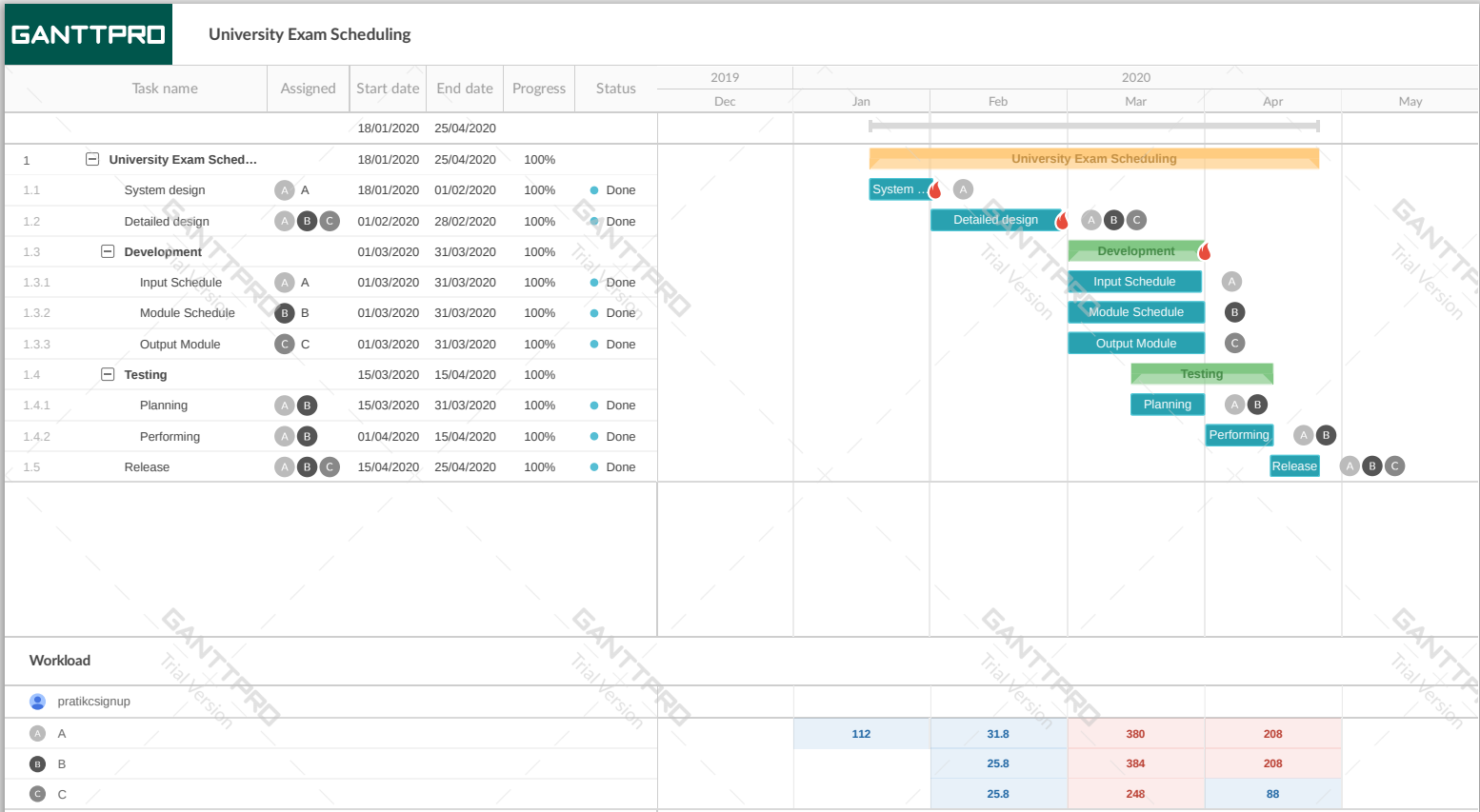
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | Coding Input module | 8 | Mar 1 | Mar 31 | A |  |
| 4 | Coding Sched module | 8 | Mar 1 | Mar 31 | B |  |
| 5 | Coding Output module | 6 | Mar 1 | Mar 31 | C |  |
| 6 | Test planning | 3 | Mar 15 | Mar 31 | A, B |  |
| 7 | Testing and integration | 5 | Apr 1 | Apr 15 | A, B |  |
| 8 | Rework and final | 3 | Apr 15 | Apr 25 | A, B, C |  |
| 9 |  |  |  |  |  |  |

The total estimated effort in person-days is: **53**

1. **Team Organization**We will have a small team of three persons A, B, and C. We use a flat team structure of peers, with one person having an additional role of project manager. As C has less time available for the project, work assigned to him is less.  
   The assignment of tasks to them will be maintained in the detailed schedule, a high-level view of which is given above.
2. **Hardware and Software resources required**The only hardware resource required is a workstation with python compiler.
3. **Quality Plan**The quality control process for this project will consist of the following:
   * 1. *SRS Review*: The SRS will be reviewed by a team.
     2. *Design Review:* Design document will be reviewed by the project team.
     3. *Unit Testing:* Each programmer is responsible for Unit Testing his module.
     4. *System Testing:* Will be done according to the system test plan, which will be reviewed.
4. **Risk Management Plan**There are no risks with this project that might need any explicit mitigation.
5. **Project Tracking**Three basic methods will be used for monitoring – project logs, weekly meetings, and reviews. As there is no timesheet system, each project member will record his activity in a project notebook and report the hours for each activity in the meetings.  
     
   Reviews will be held as per the quality plan.

**Conclusion**

Hence, we were able to using a Gantt Chart Designing Tool, design a Gantt Chart which helped us to map our requirements, set deadlines and quickly complete the project with minimal issues on time



|  |  |
| --- | --- |
| **Practical No** | 05 |
| **Title** | Risk Management-RMMM Plan |
| **Date Of Performance** |  |
| **Date Of Submission** |  |
| **Roll No** | 8322 |
| **Name Of The Student** | CHOWDHURY PRATIK VINAYAK |

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| 1 | Work Breakdown Structure(4M) |  |
| 2 | Scheduling(3M) |  |
| 3 | Resource Allocation(3M) |  |
| 5 | Total (10M) |  |

**Aim**

Develop a risk table for a University Exam Scheduling

**Description**

A **risk table** is a list of all the **risks** that could affect your software project. A **risk** is an event that is not guaranteed to happen (i.e. not 100%) that if **triggered** would **affect** your project positively or negatively. At most times, when people discuss risks in software projects, they are assuming that the risk is **negative**. If the risk event is **triggered**, i.e. comes to pass, then there is a **severity** associated with that event. Risks severity is typically low, medium, high, or catastrophic. You may have a strategy that would **mitigate** the risk. Mitigating strategies are invoked after a risk has **occurred** to reduce the severity of the outcome.

The risk table will at least list the following for each row:

1. Risk description
2. Probability
3. Severity
4. Mitigation strategies
5. Strategies to reduce the probability

Example of one line of the risk table:

* Description: Chief architect quits during project development
* Probability: Low
* Severity: Catastrophic
* Mitigation strategy:

Identify developer with best architecture skills to work with chief architect

Identify recruiters who can find qualified architects quickly

* + Strategy to reduce probability of trigger
    1. Make sure chief architect is compensated correctly
    2. Make sure that the architect has good working conditions

Typical risks include:

* Schedule risk
* Key personnel risk
* Requirements risk (i.e. that the requirements are incomplete or inconsistent)
* Learning curve risk (i.e. that your resources learn new things slower than expected)
* Technical risk

A risk table provides a project manager with a simple technique for risk projection.

A sample risk table is illustrated in the next page, Figure 1.

Impact values:

**l- Catastrophic**

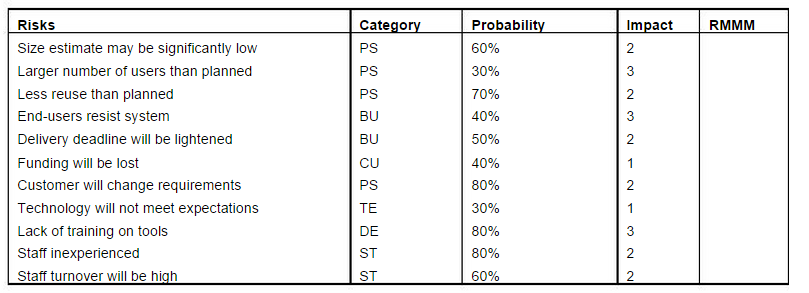
**2- Critical**

**3- marginal**

**4- Negligible**

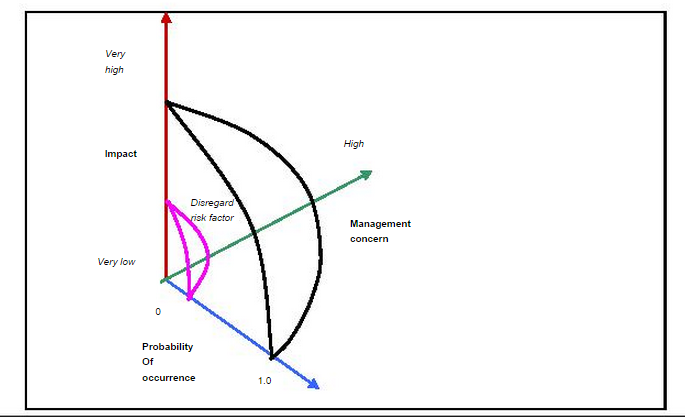
A project team begins by listing all risks (no matter how remote) in the first column of the table. This can be accomplished with the help of the risk item check-lists given earlier. Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk). The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. Individual team members are polled in round-robin fashion until their assessment of risk probability begins to converge.

**Figure 1:Sample risk table**



Next, the impact of each risk is assessed. Each risk component is assessed using the characterization presented in the sample risk table, and an impact category is determined. The categories for each of the four risk components - performance, support, cost, and schedule - are averaged to determine an overall impact value.

**Figure 2:**Risk and management concern

****

Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact. High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom. This accomplishes first order risk prioritization. The project manager studies the resultant sorted table and defines a cut-off line. The cut-off line (drawn horizontally at some point in the table) implies that only risks that lie above the line will begiven further attention. Risks that fall below the line are re-evaluated to accomplish second-order prioritization.

Referring to Figure 2, risk impact and probability have a distinct influence on management concern. A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time. However, high-impact risks with moderate to high probability and low-

impact risks with high probability should be carried forward into the risk analysis steps that follow.

All risks that lie above the cut-off line must be managed. The column labelled RMMM contains a pointer into Risk Mitigation, Monitoring and Management Plan or alternatively, a collection of risk information sheets developed for all risks that lie above the cut-off.

**Conclusion**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Risks** | **Category** | **Probability** | **Impact** | **RMMM** |
| Clients may not want to utilize system | BU | 48% | 3 | 1.44 |
| Lightened Deadlines | BU | 59% | 2 | 1.18 |
| Reduced Funding | CU | 41% | 1 | 0.41 |
| Lack of Training | DE | 86% | 3 | 2.58 |
| Project Size Estimate maybe wrong | PS | 64% | 2 | 1.28 |
| More Users than Planned | PS | 31% | 3 | 0.93 |
| Changing Requirements | PS | 85% | 2 | 1.7 |
| Inexperienced Staff | ST | 89% | 2 | 1.78 |
| May not meet Expectations | TE | 32% | 1 | 0.32 |

|  |  |
| --- | --- |
| **Practical No** | 6 |
| **Title** | General Test-Driven development |
| **Date Of Performance** |  |
| **Date Of Submission** |  |
| **Roll No** | 8322 |
| **Name Of The Student** | CHOWDHURY PRATIK VINAYAK |

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| 1 | Identification of Test Data (4 M) |  |
| 2 | Documentation of Test Case (3M) |  |
| 3 | Implementation of Test Case (3M) |  |
| 5 | Total (10M) |  |

**Aim:**

General test-driven development for a University Exam Scheduling

**Theory:**

**Test-driven development** (**TDD**) is a [software development process](https://en.wikipedia.org/wiki/Software_development_process) that relies on the repetition of a very short development cycle: requirements are turned into very specific [test cases](https://en.wikipedia.org/wiki/Test_case), then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that is not proven to meet requirements.

General Test-driven development starts with developing General Test for each one of the features. The General Test might fail as the General Tests are developed even before the development. Development team then develops and refactors the code to pass the General Test.

General Test-driven development is related to the General Test-first programming evolved as part of extreme programming concepts.

**General Test-Driven Development Process:**

* Add a General Test
* Run all General Tests and see if the new one fails
* Write some code
* Run General Tests and Refactor code
* Repeat
  1. **Steps involved:**
  2. **1. Add a General Test**

In General Test-driven development, each new feature begins with writing a General Test. To write a General Test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through [use cases a](http://en.wikipedia.org/wiki/Use_case)nd [user stories](http://en.wikipedia.org/wiki/User_story) to cover the requirements and exception conditions, and can write the General Test in whatever General Testing framework is appropriate to the software environment. It could be a modified version of an existing General Test. This is a differentiating feature of General Test-driven development versus writing unit General Tests *after* the [code](http://en.wikipedia.org/wiki/Source_code) is written: it makes the developer focus on the requirements *before* writing the code, a subtle but important difference.

* + 1. **2. Run all General Tests and see if the new one fails**

This validates that the [General Test harness](http://en.wikipedia.org/wiki/Test_harness) is working correctly, that the new General Test does not mistakenly pass without requiring any new code, and that the required feature does not already exist. This step also General Tests the General Test itself, in the negative: it rules out the possibility that the new General Test always passes, and therefore is worthless. The new General Test should also fail for the expected reason. This step increases the developer's confidence that the unit General Test is General Testing the correct constraint, and passes only in intended cases.

* + 1. **3. Write code**

The next step is to write some code that causes the General Test to pass. The new code written at this stage is not perfect and may, for example, pass the General Test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.

At this point, the only purpose of the written code is to pass the General Test; no further (and therefore unGeneral Tested) functionality should be predicted nor 'allowed for' at any stage.

* + 1. **4. Run General Tests**

If all General Test cases now pass, the programmer can be confident that the new code meets the General Test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

* + 1. **5. Refactor code**

The growing code base must be [cleaned up](http://en.wikipedia.org/wiki/Code_refactoring) regularly during General Test-driven development. New code can be moved from where it was convenient for passing a General Test to where it more logically belongs. [Duplication](http://en.wikipedia.org/wiki/Duplicate_code) must be removed. [Object](http://en.wikipedia.org/wiki/Object_%28computer_science%29), [class](http://en.wikipedia.org/wiki/Class_%28computer_programming%29), [module](http://en.wikipedia.org/wiki/Modular_programming), [variable](http://en.wikipedia.org/wiki/Variable_%28computer_science%29) and [method](http://en.wikipedia.org/wiki/Method_%28computer_programming%29) names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve [readability](http://en.wikipedia.org/wiki/Computer_programming#Readability_of_source_code) and [maintainability](http://en.wikipedia.org/wiki/Software_maintenance), which will be increasingly valuable later in the [software lifecycle](http://en.wikipedia.org/wiki/Software_lifecycle). [Inheritance hierarchies](http://en.wikipedia.org/wiki/Inheritance_%28object-oriented_programming%29) may be rearranged to be more logical and helpful, and perhaps to benefit from recognised [design patterns](http://en.wikipedia.org/wiki/Software_design_pattern). There are specific and general guidelines for refactoring and for creating clean code.[[6]](http://en.wikipedia.org/wiki/Test-driven_development#cite_note-6)[[7]](http://en.wikipedia.org/wiki/Test-driven_development#cite_note-7) By continually re-running the General Test cases throughout each refactoring phase, the developer can be confident that process is not altering any existing functionality.

The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to the removal of any duplication between the General Test code and the production code—for example [magic numbers or strings](http://en.wikipedia.org/wiki/Magic_number_%28programming%29) repeated in both to make the General Test pass in Step 3.

* + 1. **6. Repeat**

Starting with another new General Test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each General Test run. If new code does not rapidly satisfy a new General Test, or other General Tests fail unexpectedly, the programmer should [undo](http://en.wikipedia.org/wiki/Undo) or revert in preference to excessive [debugging](http://en.wikipedia.org/wiki/Debugging). [Continuous integration](http://en.wikipedia.org/wiki/Continuous_integration) helps by providing revertible checkpoints. When using external libraries it is important not to make increments that are so small as to be effectively merely General Testing the library itself,[[4]](http://en.wikipedia.org/wiki/Test-driven_development#cite_note-Newkirk-4) unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the software under development.

* 1. **Development style**

To achieve some advanced design concept such as a [design pattern](http://en.wikipedia.org/wiki/Design_pattern), General Tests are written that generate that design. The code may remain simpler than the target pattern, but still pass all required General Tests. This can be unsettling at first but it allows the developer to focus only on what is important.

Writing the General Tests first: The General Tests should be written before the functionality that is to be General Tested. This has been claimed to have many benefits. It helps ensure that the application is written for General Testability, as the developers must consider how to General Test the application from the outset rather than adding it later. It also ensures that General Tests for every feature get written. Additionally, writing the General Tests first leads to a deeper and earlier understanding of the product requirements, ensures the effectiveness of the General Test code, and maintains a continual focus on [software quality](http://en.wikipedia.org/wiki/Software_quality).When writing feature-first code, there is a tendency by developers and organizations to push the developer on to the next feature, even neglecting General Testing entirely. The first TDD General Test might not even compile at first, because the classes and methods it requires may not yet exist. Nevertheless, that first General Test functions as the beginning of an executable specification.

Each General Test case fails initially: This ensures that the General Test really works and can catch an error. Once this is shown, the underlying functionality can be implemented. This has led to the "General Test-driven development", which is "red/green/refactor," where red means *fail* and green means *pass*. General Test-driven development constantly repeats the steps of adding General Test cases that fail, passing them, and refactoring. Receiving the expected General Test results at each stage reinforces the developer's mental model of the code, boosts confidence and increases productivity.

General Test-driven development, revolves around a short iterative development cycle that goes something like this:

* Before writing any code, you must first write an automated General Test for your code. While writing the automated General Tests, you must take into account all possible inputs, errors, and outputs. This way, your mind is not clouded by any code that's already been written.
* The first time you run your automated General Test, the General Test should fail—indicating that the code is not yet ready.
* Afterward, you can begin programming. Since there's already an automated General Test, as long as the code fails it, it means that it's still not ready. The code can be fixed until it passes all assertions.
* Once the code passes the General Test, you can then begin cleaning it up, via refactoring. As long as the code still passes the General Test, it means that it still works. You no longer have to worry about changes that introduce new bugs.
* Start the whole thing over again with some other method or program.

**Benefits of TDD:**

1. Much less debug time
2. Code proven to meet requirements
3. General Tests become Safety Net
4. Near zero defects
5. Shorter development cycles

**Implementation:**

**Problem Statement: University Exam Scheduling**

Here we specify all the test cases to be used for system testing. These test cases form a part of the system testing plan. For test case specifications we specify the different conditions that need to be tested, along with the test cases used for testing those conditions and the expected outputs. Then the data files used for testing are given. The test cases are specified with respect to these data files. The test cases have been selected using the functional approach. The goal is to test the different functional requirements, as specified in the requirements document. Test cases have been selected for both valid and invalid inputs.

**Conclusion:**

Based on above General Testing Ideas, we have created the following Unit Tests

SEQ TEST CASE CONDITION BEING EXPECTED OUTPUT

NO. [File] CHECKED

1. Empty file Empty F1 Print message and stop
2. Empty file Empty F2 Print message and stop
3. No file F1 Does not exist Print message and stop
4. No file F2 Does not exist Print message and stop

|  |  |
| --- | --- |
| **Practical No** | 7 |
| **Title** | Black Box Testing |
| **Date Of Performance** |  |
| **Date Of Submission** |  |
| **Roll No** | 8322 |
| **Name Of The Student** | CHOWDHURY PRATIK VINAYAK |

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| 1 | Accurate Flow Graph (3M) |  |
| 2 | Computation of Cyclomatic Complexity (3M) |  |
| 3 | Determination of basis set(2M) |  |
| 4 | Determination of test data(2M) |  |
| 5 | Total (10M) |  |

**Aim**

To design test cases for performing black box testing (equivalence partitioning and boundary value analysis) for University Exam Scheduling.

**Description**Black box testing (also called behavioral/functional testing), focuses on the functional requirements of the software. The test designer selects valid and invalid inputs and determines the correct output. There is no knowledge of the test object's internal structure. It is not an alternative to white box testing, and a complementary approach to white box testing. It uncovers different class of errors than white box testing.

Black-box testing attempts to find errors in the following categories

Incorrect/missing function.

Interface errors.

Errors in database/external database access.

Behavior/performance errors.

Black Box testing techniques are:

***1. Equivalence Partitioning***

It is the black-box technique that divides the input domain into classes of data from which test cases can be derived. Equivalence partitioning defines test cases that uncover classes of errors thereby reducing the no. of test cases that must be developed. If the input condition specifies a range, one valid and two invalid equivalence classes are defined. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined. If an input condition specifies a member of a set, one valid and one invalid equivalence class is defined. If an input condition is boolean, one valid and one invalid equivalence class is defined. By applying these guidelines, test cases for each input domain can be developed.

E.g. A program reads an input number in the range 1 and 5000 and computes the square of the input number. 3 Equivalence classes are:

* Set of numbers less than 1.
* Set of numbers between 1 and 5000.
* Set of numbers greater than 5000.

A possible test case is {-10,100,7000}

***2. Boundary Value Analysis***

It focuses on the boundaries of the input domain rather than its center. The following guidelines can be used for performing boundary value analysis:

* 1. If the input condition specifies a range bounded by values a and b, test cases should include a and b, values just above and just below a and b
  2. If an input condition specifies and number of values, test cases should exercise the minimum and maximum numbers, as well as values just above and just below the minimum and maximum values
  3. Apply guidelines 1 and 2 to output conditions, test cases should be designed to produce the minimum and maximum output reports
  4. If internal program data structures have boundaries (e.g. size limitations), be certain to test the boundaries

**Conclusion**

Hence, based on what we learnt from this experiment; we were able to understand how to perform Black Box Testing on our project.

The tests used are mentioned below

SEQ TEST CASE CONDITION BEING EXPECTED OUTPUT

NO. [File] CHECKED

For checking FILE1 format error

|  |  |  |  |
| --- | --- | --- | --- |
| 5 | [F1.1] | Incorrect course no. | Print course no. and |
|  |  | format | error message |
| 6 | [F1.7] | More than allowed (30) | Error message and skip |
|  |  | courses | to lecture times |
| 7 | [F1.4] | Course list empty | Error message and skip |
|  |  | to lecture times |  |
| 8 | [F1.5] | Spelling of header | Error message and stop |
| 9 | [F1.1] | Lecture time format | Print time, error message, |
|  |  |  | and continue |
| 10 | [F1.2] | More than allowed no. | Error message, discard |
|  |  | of lecture times (15) | extra and skip to room no.s |
| 11 | [F1.4] | Lecture times list empty | Print “No lecture times” |
|  |  |  | and parse rooms |
| 12 | [F1.1] | Incorrect room no format | Print room no. and message |
| 13 | [F1.1] | No colon (:) between | Continue |
|  |  | room# and capacity |  |
| 14 | [F1.1] | Capacity format | Print message with room no. and |
|  |  |  | capacity and continue |
| 15 | [F1.1] | Capacity more than | Error message, continue |
|  |  | limit (300) |  |
| 16 | [F1.1] | Capacity less than 10 | Error message, continue |
| 17 | [F1.7] | More than 20 room#, cap | Error message, stop |
|  |  | entries |  |
| 18 | [F1.4] | Room list empty | Error message, stop |
| 19 | [F1.1] | No correct room entries | Error message, no schedu- |
|  |  |  | ling, continue parsing |
| 20 | [F1.3] | Same course no entered | Print message and discard |
|  |  | more than once | the entry |
| 21 | [F1.3] | Duplicate lecture time | Print message, discard |
|  |  |  | it, and continue |
| 22 | [F1.3] | Duplicate room entry | Print message, ignore |
|  |  |  | it, and continue |

SEQ TEST CASE CONDITION BEING EXPECTED OUTPUT

NO. [File] CHECKED

FILE2 format (for FILE1, F1.8 is used)

|  |  |  |  |
| --- | --- | --- | --- |
| 23 | [F2.1] | Enrollment *≤* 2 | Print message, ignore  it, and continue |
| 24 | [F2.1] | Enrollment in range | Executes normally |
|  |  | [3–250] |  |
| 25 | [F2.1] | Enrollment exceeds 250 | Print message,continue |
| 26 | [F2.2] | No preference specified | Scheduled |
| 27 | [F2.1] | More than allowed number | Print message and |
|  |  | of preferences (5). | discard the rest |
| 28 | [F2.1] | Duplicate course entry | Print message and |
|  |  |  | ignore duplicate |

Consistency of FILE2 with FILE1. File F1.8 used for file 1.

|  |  |  |  |
| --- | --- | --- | --- |
| 29 | [F2.1] | Course not present in | Print message, ignore |
|  |  | the list of offered courses | it, and continue |
| 30 | [F2.1] | Preference not found in | Print message and |
|  |  | lecture time list | ignore the preference |
| 31 | [F2.1] | Enrollment *>* max. room | Error message |
|  |  | capacity available |  |
| 32 | [F2.4] | Missing enrollment field | Ignore the course |

SCHEDULING cases. File F1.8 used for file 1.

|  |  |  |  |
| --- | --- | --- | --- |
| 33 | [F2.4] | No valid courses in F2 | Print message and stop |
| 34 | [F2.2] | No PG course with prefs | Schedule |
| 35 | [F2.4] | No UG course with prefs | Schedule |
| 36 | [F2.4] | No PG courses with no pref | Schedule |
| 37 | [F2.2] | No two courses allotted | The first course is given |
|  |  | at the same time and in | the first preference |
|  |  | the same room |  |
| 38 | [F2.2] | Room capacity is more | Course scheduled in a |
|  |  | than the classroom | room with capacity |
|  |  |  | more than enrollment |
| 39 | [F2.2] | PG courses given priority | PG course is scheduled; |
|  |  | over UG courses even if | UG course faces conflict |
|  |  | UG course appears before |  |
|  |  | the PG course in input |  |
| 40 | [F2.2] | Courses scheduled in the | The first course is given |
|  |  | order they appear in the | the best pref., second the |
|  |  | input file | next pref., and so on |
| 41 | [F2.2] | Highest possible preference | The nth pref. honored with |
|  |  | of a course is honored | explanation for all the |
|  |  |  | earlier n-1 preferences |

|  |  |  |  |
| --- | --- | --- | --- |
| SEQ  NO. | TEST CASE  [File] | CONDITION BEING  CHECKED | EXPECTED OUTPUT |
| 42 | [F2.3] | No two PG courses | The first one scheduled |
|  |  | scheduled in the same slot | and conflict shown for |
|  |  | even if same pref. given | the second course |
| 43 | [F2.3] | PG course with pref. given | Courses with pref.s are |
|  |  | priority over PG courses | scheduled before |
|  |  | with no preference |  |
| 44 | [F2.2] | PG courses with no pref.s | PG course scheduled |
|  |  | are guaranteed a room even | and conflict generated |
|  |  | if some UG course has to | for the UG course |
|  |  | be “unscheduled” |  |
| 45 | [F2.2] | No room with required | Error message |
|  |  | capacity available for UG |  |
|  |  | course with no preference |  |

|  |  |
| --- | --- |
| **Practical No** | 8 |
| **Title** | White box Testing |
| **Date Of Performance** |  |
| **Date Of Submission** |  |
| **Roll No** | 8322 |
| **Name Of The Student** | CHOWDHURY PRATIK VINAYAK |

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| 1 | On Time Completion(2M) |  |
| 2 | Completeness(4M) |  |
| 3 | Correctness(4M) |  |
| 5 | Total (10M) |  |

**Aim**

To design test cases for performing white box testing for University Exam Scheduling.

**Description**

White box testing, also called glass box testing is a testing technique which exercises the internal logic of software components. Using white box testing, the software engineer can derive test cases that

1. Guarantee that all independent paths within a module have been exercised at least once.
2. Exercise all logical decisions on their true and false sides.
3. Execute all loops at their boundaries
4. Exercise internal data structures to ensure their validity.

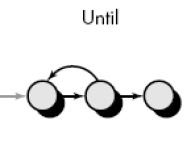
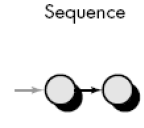
Different types of White Box Testing techniques are:

1. ***Basis Path Testing***

*Basis path testing* is a white-box testing technique first proposed by Tom McCabe. Basis path method enables to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test Cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing. The flow graph depicts logical control flow within a program. The following steps constitute basis path testing.

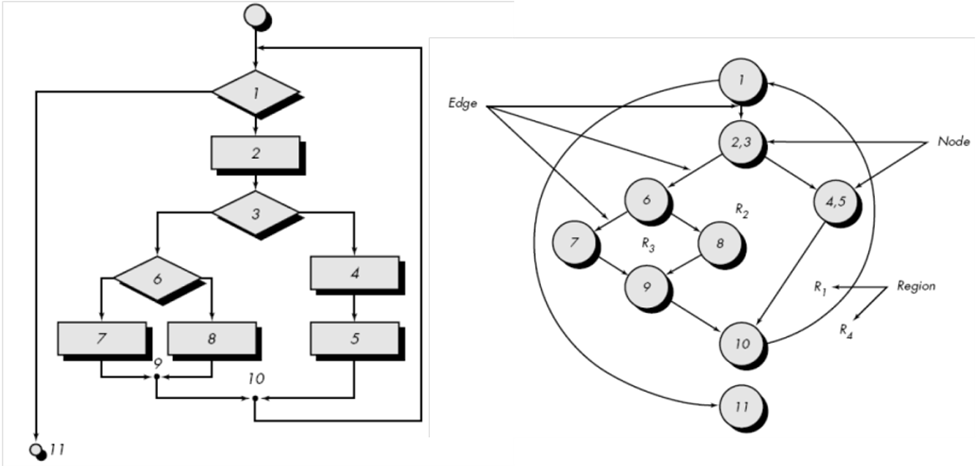
**Step1: Map flowchart into a corresponding flow graph**

* Each circle, called a flow graph node, which represents one or more procedural statements.
* A sequence of process boxes and a decision diamond can map into a single node.
* The arrows on the flow graph, called *edges* or *links,* represent flow of control and are analogous to flowchart arrows.
* Areas bounded by edges and nodes are called regions.



**Figure 8.1 Flow graph notations**

* When counting regions, we include the area outside the graph as a region.
* Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.



**Figure 8.2 Example flowchart and corresponding flow graph**

**Step 2 : Compute the cyclomatic complexity CC**

* *Cyclomatic complexity* is a software metric that provides a quantitative measure of the logical complexity of a program.
* The value computed for cyclomatic complexity defines the number of independent paths in the program.
* Hence gives the number of test cases to ensure that all statements have been executed at least once.
* An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition.

**Complexity is computed in one of three ways:**

1. The regions formed in the flow graph correspond to cyclomatic complexity.
2. Cyclomatic complexity, *V*(*G*), for a flow graph, *G,* is defined as

*V*(*G*) = *E* - *N* + 2

where *E* is the number of flow graph edges, *N* is the number of flow graph nodes.

1. Cyclomatic complexity, *V*(*G*), for a flow graph, *G,* is also defined as

*V*(*G*) = *P* + 1

where *P* is the number of predicate nodes contained in the flow graph G.

For the example in Figure 8.2 above,

1. The flow graph has four regions.
2. *V*(*G*) = 11 edges - 9 nodes + 2 = 4.
3. *V*(*G*) = 3 predicate nodes + 1 = 4.

**Step 3: Determine a basis set (set of independent paths)**

1. path 1: 1-11
2. path 2: 1-2-3-4-5-10-1-11
3. path 3: 1-2-3-6-8-9-10-1-11
4. path 4: 1-2-3-6-7-9-10-1-11

**Step 4: Prepare test cases that will force execution through each of the basis paths**

1. ***Control Structure Testing***

Control structure testing is more comprehensive than basis path testing. This method uses different categories of tests that are listed below.

1. Condition testing (e.g. branch testing) focuses on testing each decision statement in a software module. It is important to ensure coverage of all logical combinations of data that may be processed by the module (a truth table may be helpful).
2. Data flow testing selects test paths based according to the locations of variable definitions and uses in the program (e.g. definition use chains).
3. Loop testing focuses on the validity of the program loop constructs (i.e. while, for, go to).
4. **Condition testing**

Conditional testing strategy can be applied to the different types of conditions that are possible which are listed below.

* Compound conditions

2 or more simple conditions connected with AND, OR.

Eg : (a>b) AND (c<d)

* Relational expression

(E1 rel-op E2) ;E1, E2 --à arithmetic expr.

Eg : (a\*b+c) > (a+b+c)

* Boolean expression

(B1 AND B2)

Condition testing strategies are described below.

1. **Branch Testing**

* It is the simplest condition testing strategy.
* For a compound condition C, the true and false branches of C and every simple condition need to be executed at least once.
* Eg : if ( a>0 && b== null )

condition coverage can be achieved by testing with

a=1 b != null

a=1 b = null

a=0 b != null

a=0 b = null

1. **Domain testing**

* For a relational expression, 3 tests are required:

Eg : E1<Rel-op>E2

3 test cases are:

1. E1<E2

2. E1>E2

3. E1=E2

If relational operator is incorrect , all 3 tests guarantee detection of relational operator error.

1. **Branch and relational operator testing**

* Branch and Relational Operator Testing – uses condition constraints

Example 1:

**C1: B1 & B2**

Where B1, B2 are boolean conditions

Condition constraint of form D1, D2 where D1 and D2 can be true (t) or false (f)

The branch and relational operator test requires the constraint set { (t,t), (f,t), (t,f) } to be covered by the execution of C1

Example 2:

**C2: B1 & (E3=E4)**

B1 - boolean expr

E3, E4 - arithmetic expr

Condition constraint of C2 is (D1, D2)

D1- (t/f)

D2- >,=,<

Hence the constraint set for C2 is

(t,t)------à (t,=)

(t,f)----à (t,<) and (t,>)

(f,t)---à (f,=)

Example 3:

**C3: (E1>E2) & (E3=E4)**

E1,E2,E3,E4 ---arithmetic expr.

Constraint set for C3----à

{ (>,=) (=,=) (<,=) (>,>) (>,<) }

(t,t) (f,t) (f,t) (t,f) (t,f)

1. **Data Flow testing**

* Data flow testing selects test paths of a program according to the locations of definitions and uses of variables in the program.
* A USE is a reference to a variable’s value. A DEF is an assignment of a new value to the variable.
* A DEF-USE pair (DU chain) is a path from the point the variable is defined to the point the variable is referenced. Data flow testing requires that all DEF-USE pairs to be executed.

Example 1

if(some\_exp) //1

some\_var=1; //2

else //3

some\_var=2; //4

If(some\_case) //5

P1(some\_var); //6

else //7

P2(some\_var); //8

1. DEF: 2 and 4

USE : 6 and 8

4 DEF-USE pairs ----à

DEF-USE-Paths DEF USE PATH

1 2 6 <1-2-5-6>

2 2 8 <1-2-5-8>

3 4 6 <1-4-5-6>

4 4 8 <1-4-5-8>

Test Cases :

(some\_exp=true, some\_case=true) path 1

(some\_exp=true, some\_case=false) path 2

(some\_exp=false, some\_case=true) path 3

(some\_exp=false, some\_case=false) path 4

Example 2



**Figure 8.3 Example for Data Flow Testing**

* DU Chains of the Odd() Example

– (x, 3, 4), (x, 3, 5), (x, 3, 7)

– (x, 5, 8), (x, 7, 8)

NOTE : (x, 3, 8) is NOT a DU chain since the value of x at Line 3 is redefined at Lines 5 and 7 before it reaches the use at Line 8

Test paths selected according to all use coverage:

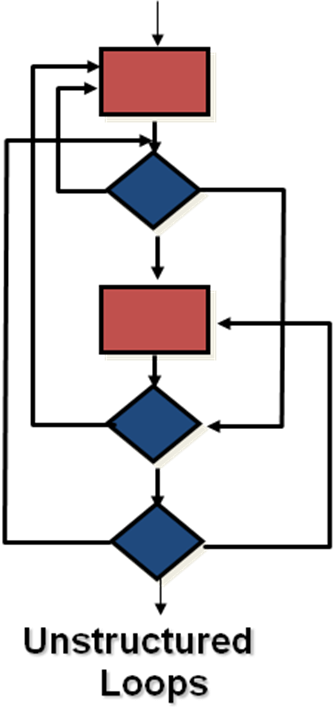
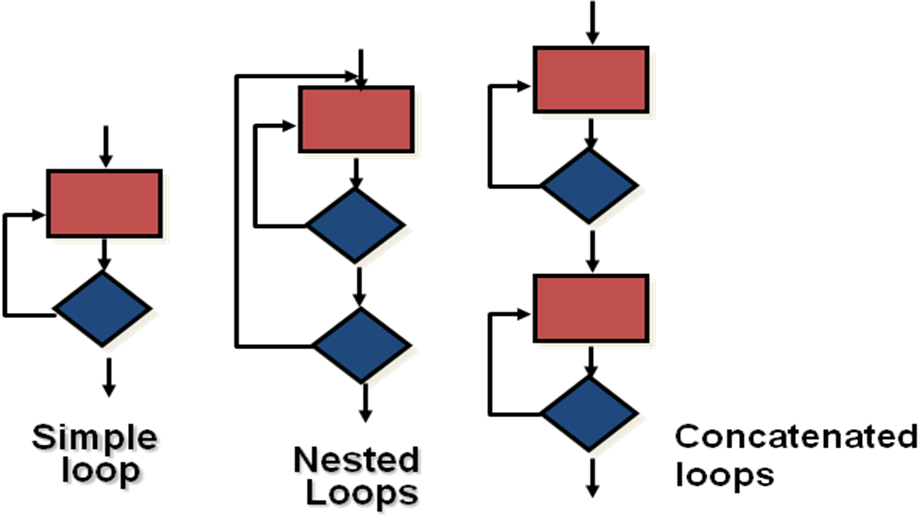
path1 1-2-3-4-7-8 cover (x, 3, 4), (x, 3, 7), (x, 7, 8)

path2 1-2-3-4-5-8 cover (x, 3, 4), (x, 3, 5), (x, 5, 8)

1. **Loop Testing**

Loop Testing is a white box testing technique that focuses exclusively on the validity of loop constructs. Four classes of loops can be defined:

* Simple loops
* Concatenated loops
* Nested loops
* Unstructured loops.

******

**Figure 8.4 Different types of loops**

1. **Testing simple loops**

The following sets of tests can be applied to simple loops, where ‘n’ is the maximum number of allowable passes through the loop.  
  
1. Skip the loop entirely.  
  
2. Only one pass through the loop.  
  
3. Two passes through the loop.  
  
4. ‘m’ passes through the loop where m is less than n.  
  
5. n-1, n, n+1 passes through the loop.

1. **Nested Loops**

If we extend the test approach from simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases.  
  
1. Start at the innermost loop. Set all other loops to minimum values.  
  
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values.

3. Work outward, conducting tests for the next loop, but keep all other outer loops at minimum values and other nested loops to "typical" values.  
  
4. Continue until all loops have been tested.

1. **Concatenated Loops**

Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other.

However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. In this case, treat them as nested loops and perform testing.

1. **Unstructured Loops**

Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

**Conclusion**

Hence, based on what we learnt from this experiment; we were able to understand how to perform White Box Testing on our project.

The tests used are mentioned below

* + - “Process File 1” component maps to “Validate File 1” module and its subtree.

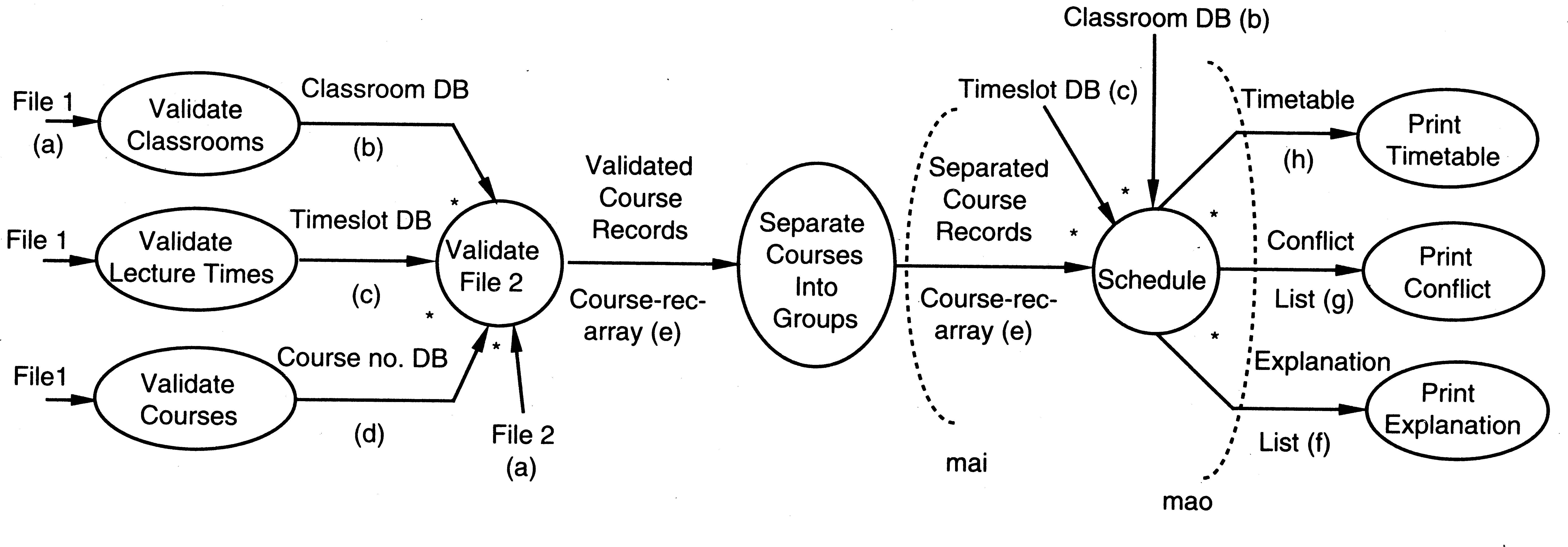


Figure 1: Data flow diagram for the case study

“Process File 2” component maps to “Validate File 2” and “Separate Courses” modules (and their subtrees).

•

“Schedule the courses” components map to the “Schedule” and “Output” modules (and their subtrees)

•

The pipes in the architecture are implemented in this design by the parameter passing mechanism.

•

**Cyclomatic Complexity**

***M = E – N + 2P***

*where,  
E = the number of edges in the control flow graph  
N = the number of nodes in the control flow graph  
P = the number of connected components*

Therefore Cyclomatic Complexity = E – N + 2P

= 9 – 14 + 2(8)

= 11

|  |  |
| --- | --- |
| **Practical No** | 09 |
| **Title** | Cohesion & Coupling |
| **Date Of Performance** |  |
| **Date Of Submission** |  |
| **Roll No** | 8322 |
| **Name Of The Student** | CHOWDHURY PRATIK VINAYAK |

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| 1 | On Time Completion(2M) |  |
| 2 | Completeness(4M) |  |
| 3 | Correctness(4M) |  |
| 5 | Total (10M) |  |

**Aim**

Draw Architecture diagram and incorporate Cohesion and Coupling for each module of University Exam Scheduling

**Description**

The aim of performing this experiment is to implement data flow architecture in your project and show type of cohesion between operations and coupling between components in your project.

For Good project design, Cohesion should be high and coupling should be s low as possible.

**Coupling**

* The degree of interdependence between two modules”
* We aim to minimize coupling - to make modules as independent as possible

**Types of Coupling**

High Cohesion

Data coupling

Stamp coupling

Control coupling

Hybrid coupling

Common coupling

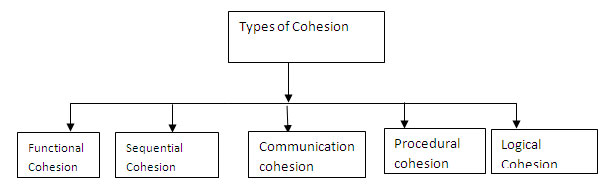
Content coupling

* **Data Coupling**Modules communicate by parameters
* **Data coupling problems**Too many parameters - makes the interface difficult to understand and possible error to occur  
  A composite data is passed between modules
* **Control coupling**A module controls the logic of another module through the parameter
* **Hybrid coupling**A subset of data used as control
* **Common coupling**Use of global data as communication between modules
* **Content coupling**A module refers to the inside of another module

**Cohesion**

* “The measure of the strength of functional relatedness of elements within a module”
* Elements: instructions, groups of instructions, data definition, call of another module
* Strong cohesion will reduce relations between modules - minimize coupling

**Types of Cohesion**

****

**Functional cohesion (Most Required)**All elements contribute to the execution of one and only one problem-related task

**Sequential cohesion**Elements are involved in activities such that output data from one activity becomes input data to the next

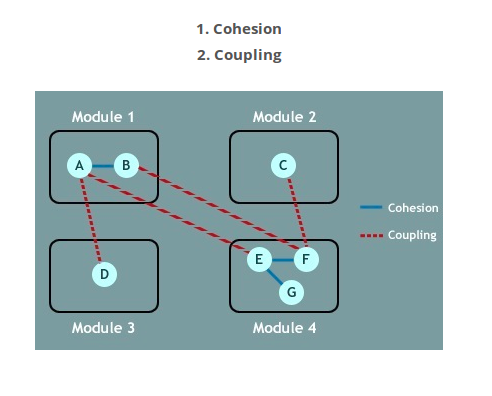
**Communicational Cohesion**  
Elements contribute to activities that use the same input or output data

**Procedural cohesion**Elements are related only by sequence, otherwise the activities are unrelated

**Temporal cohesion**Elements are involved in activities that are related in time

**Logical cohesion**Elements contribute to activities of the same general category

**Coincidental cohesion(Least Required)**Elements contribute to activities with no meaningful relationship to one another



**Conclusion**

Let us evaluate this architecture concerning some properties.

|  |  |
| --- | --- |
| **Criteria** | **Evaluation of the Proposed Architecture** |
| Change in some file  format | Good – change should impact only the filter  for that file. |
| Change in scheduling  algorithm | Good – only the scheduling component needs to be changed. |
| Adding new constraints for  scheduling | Good – only the scheduling component needs to be changed |
| Replacing files with GUI | Poor – switching to a GUI interface will require changing almost the complete implementation with this architecture. The  scheduling component can still be used. |

|  |  |
| --- | --- |
| Extension to web-based | Poor – This architecture does not support this change; will require a complete rewrite of the  system. |
| Provision of additional securities (passwords, etc) | Average – The architecture is not designed with security in mind. However, it is possible to add a security component for verification  in the start. |

|  |  |
| --- | --- |
| **Practical No** | 10 |
| **Title** | Design Pattern |
| **Date Of Performance** |  |
| **Date Of Submission** |  |
| **Roll No** | 8322 |
| **Name Of The Student** | CHOWDHURY PRATIK VINAYAK |

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| 1 | Understanding of Junit Framework(4M) |  |
| 2 | Implementation (4M) |  |
| 3 | Coding Standards(2M) |  |
| 5 | Total (10M) |  |

**Aim**

Application of at least two types of design patterns in selected case study

**Description**

1. **OVERVIEW**
   1. **System Overview**

The university exam scheduling system takes input from the Professors about their preferences, about course offerings and classrooms from the department secretary, and then proposes a suitable schedule for the courses.

* 1. **System Context**

The system context is defined clearly in the SRS. The department is the main sink of the information. The main sources of information are the Professors (who provide information about their time preferences and maximum enrollment,) and the department secretary (which provides information about courses begin offered, time slots, and available classrooms.)

* 1. **Stakeholders of PIMS**

The main stakeholders and their concerns are:

* + 1. **Professors:** Their main concern is that their highest priority should be satisfied. This means that the algorithm for scheduling should be such that it can easily be changed with a better algorithm later.
    2. **Department secretary/Head:** The schedule should be fair and should utilize the

resources well. (Again, this means that the scheduling algorithm should be upgradable.)

* 1. **Scope of this Document**

This document describes the proposed architecture for the university exam scheduling system. For architecture, we consider only the component and connector view.

* 1. **Definitions and Acronyms**

As given in the SRS.

1. **ARCHITECTURE DESIGN**

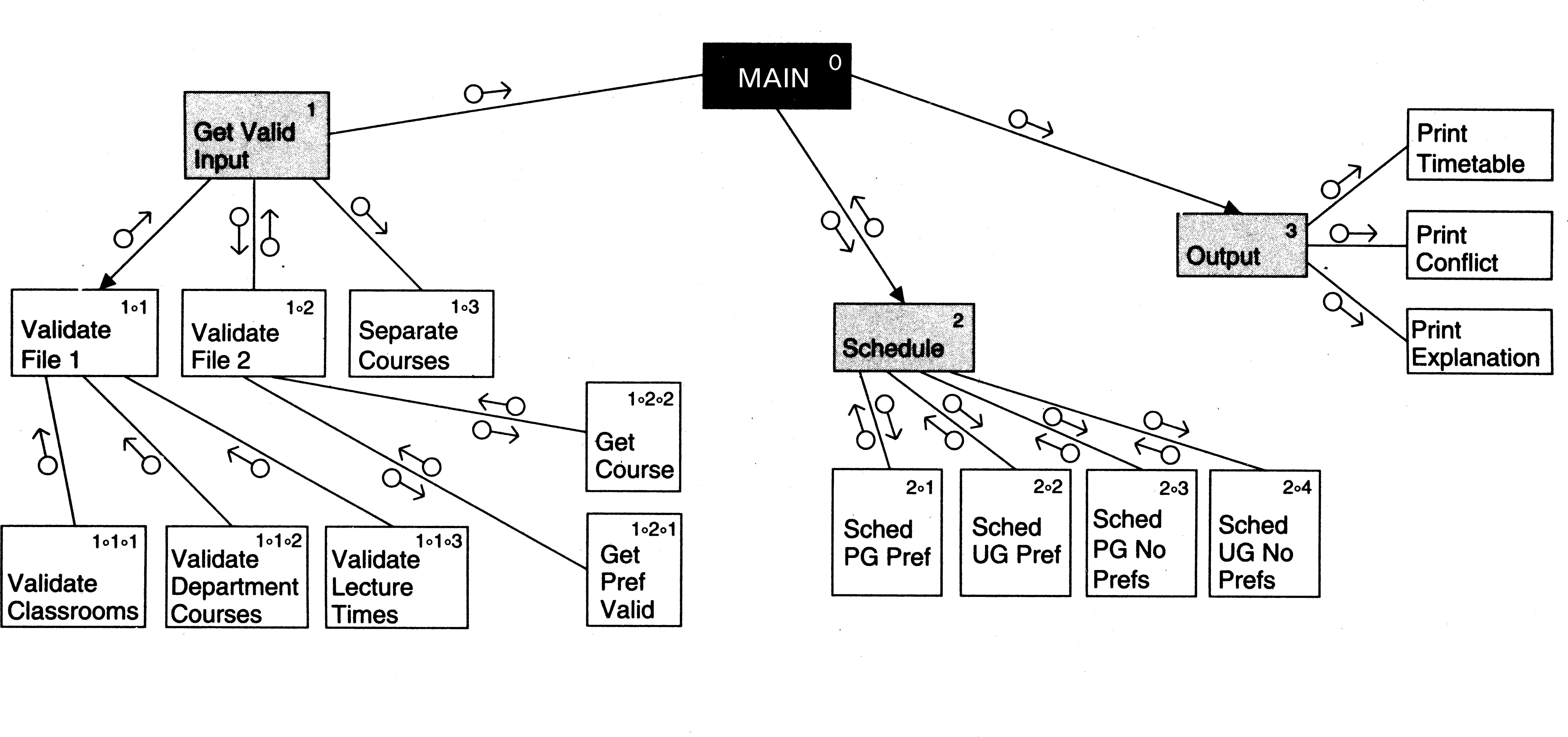
As this is a batch processing-type system with inputs coming and output being produced, the most natural style will be the pipe-and-filter style. We use this style for the architecture of the system. The proposed architecture is shown below.

This architecture has three filter components – one to process the information provided in File 1, the other to process the information provided in File 2, and the third to produce the schedule. As the information produced by the first component is also used in the processing of information from File 2 as well as for scheduling, the connections are set accordingly.

In this architecture, we do not require that the components be executed in parallel. For ease of implementation, they may be executed in sequential order, particularly since the file processing modules are not likely to consume much time – the main processing time will be needed by the third component and that remains the same in parallel or serial execution. Any (synchronous or asynchronous) method can be used to support the pipes.

**Conclusion**

Structural Design



Architecture Diagram

