Software Design Document
for
Teaching Assistant Management System

Group No. 7

Sai Ramana Reddy	CS17BTECH11022
Puneet Mangla	CS17BTECH11029
Vijay Tadikamalla	CS17BTECH11040
Tungadri Mandal	CS17BTECH11043

Contents

1 Overview

The goal of this project is to develop a system for managing TA allocation, activities and feedback, serving as an intermediary between students and professors. Different requirements have to be satisfied by the final system which are specified in the SRS.

This document specifies the final system design for this system. It also gives some explanations on how the design evolved and why some design decisions were taken.

1.1 DFD of the System

The DFD describes the various inputs, outputs and flow of data in the system. The DFD illustrates at a high level, how the system processes the data and meets the requirements. We don't include design implementation and analysis of User Interface. The DFD of our system is shown in Figure 1.1.

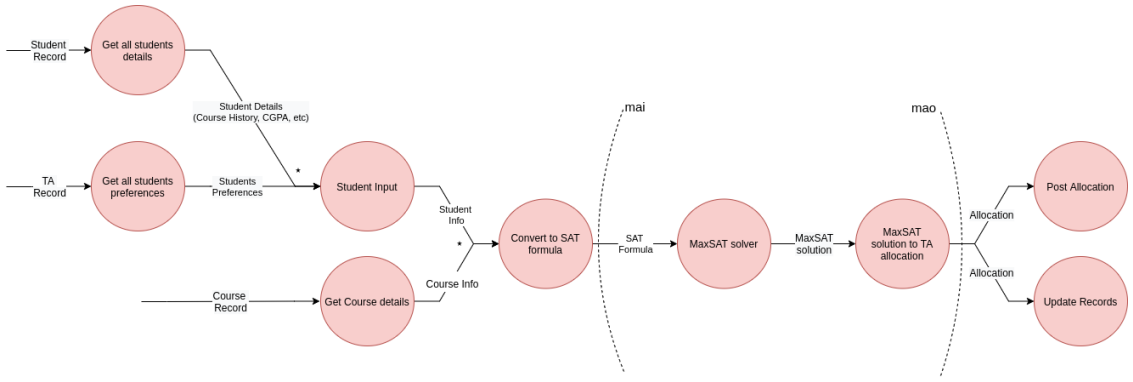

Figure 1.1: Final DFD

1.2 Structure Chart

The structure chart illustrates the various modules in the system and the interactions between them. This describes the overall structure of the design.

Note that this design is consistent with the architecture of the system given in its architecture document. Mapping the modules in this design to the components in the architecture is also quite straightforward.

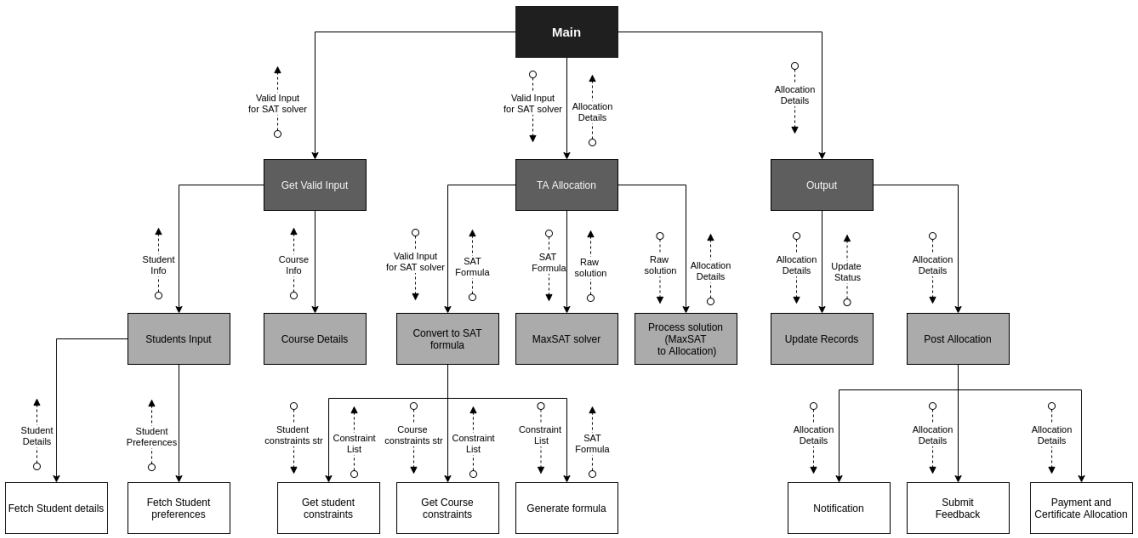

Figure 1.2: Structure Chart

1.3 Metric Analysis

Below table reports the fan-in, fan-out, LOC estimate, inflow and outflow specifications of each of the final factored modules.

Note: Here, we assume that the inflow value depends on the information flowing into the module (like the input parameters) and outflow value depends on the information flowing out of the module (like the return value)

Module	Fan-In	Fan-Out	LOC	Inflow	Outflow
`get_valid_input`	1	2	100	1	2
`get_student_constraints`	1	0	200	1	1
`get_course_constraints`	1	0	300	1	1
`ta_allocation`	1	3	50	2	1
`convert_to_sat_formula`	1	3	150	2	1
`gen_formula`	1	0	50	2	1
`process_solution`	1	0	200	1	1
`output`	1	2	50	2	1
`update_records`	1	0	100	2	1
`post_allocation`	1	3	100	2	0
`notify_allocation`	1	0	50	1	0
`submit_feedback`	1	0	100	2	0
`distribute_incentive`	1	0	150	1	0

Total expected size of your software in terms of LOC[1] : 1600 ± 500
Top-3 modules, along with counts, in terms of fan out and fan in

- Fan-In : All modules have same fan-in value which is 1.
- Fan-Out : `ta_allocation` (3), convert_to_sat_formula (3), `post_allocation` (3)

[1]Note that we don't consider User Interface designs here

1.3.1 Complexity Analysis

We know the following,

```
Error prone: If Dc > avg complexity + std_dev
Complex: If avg complexity < Dc < avg + std_dev
Normal: Otherwise
```

where $Dc = size * (inflow * outflow)^2$

Hence, the following are the most complex or error prone module in input, transformation and output subsystem.

Subsystem	Error-prone	Complex
`Input`	get_valid_input	get_course_constraints
`Transform`	convert_to_sat_formula	process_solution
`Output`	None	None

Intuitively, we can see that the top most module (get_valid_input) in the Input subsystem is the most error prone because of the large size and dependency on other modules.

In the output subsystem, most of the modules are small in size because they just perform simple functions like the notification and database updates. Hence they are neither error prone nor complex.

2 Interfaces

```
class Course:
    id -> int
    segment -> tuple
    num_ta -> int
    constraint_string -> str

class Student:
    roll_no -> str
    mail -> str
    name -> str
    year -> int
    branch -> str
    stream_type -> str
    course_history -> [ CourseHisory() ]
    preferences -> str

class AllocatedStudents:
    course -> Course()
    students -> [Students()]

class ValidInput:
    students -> [ Student() ]
    courses -> [ Course() ]

def get_valid_input(DB_URL):
    """
    Server gets all the required input from the Database
    """

    valid_input = ValidInput()
    db_manager = database_manager.get_connection(DB_URL)
    valid_input.students = get_student_info(db_manager)
    valid_input.courses = get_course_info(db_manager)
    return valid_input

def get_student_input(db_manager):
    """
    Get all student Info from Student Record and TA record
    """
    # Database query to get valid student info as a
    # input from Student Record and TA record
    db_query = "..."
    student_info = db_manager.execute_query(db_query, Student)
    return student_info

def get_course_info(db_manager):
    """
    Get all Course Details from Course Record
    """
    # Database query to get valid course info as a input from Course record
    db_query = "..."
```

```
    course_info = db_manager.execute_query(db_query, Course)
    return course_info

def ta_allocation(valid_input):
    """
    convert student preferences and course requirements to SAT
    formula and feed into solver to get solution
    """
    sat_formula = convert_to_sat_formula(valid_input.students, valid_input.courses)
    solver = MaxSATSolver(formula, **kwargs)
    raw_solution = (valid_input.courses,solver.solve())
    solution = process_solution(raw_solution)
    return solution

def convert_to_sat_formula(students, courses):
    students_constraints = [] # preferences of all students in valid constraint format
    for student in students:
        student_con = get_student_constraints(student.preference_string)
        students_constraint.append(student_con)
    course_constraints = [] # details of all courses in valid constraint format
    for course in courses:
        course_con = get_course_constraints(course.constraint_string)
        course_constraints.append(course_con)
    formula = gen_formula(student_constraints, course_constraints)
    return formula

def get_student_constraints(student_preference_str):
    constraint_list = []
    raw_constraints = student_preference_str
    constrings = student_preference_str.split(con_string_separator)
    for constring in constrings:
        c = newConstraint()
        # parse and process individual constraint string
        # filter out valid courses
        # check for hard and soft constraints
        constraint_list.append(c)
    return constraint_list

def get_course_constraints(course_constraint_str):
    constraint_list = []
    constrings = course_constraint_str.split(con_string_separator)
    for constring in constrings:
        c = newConstraint()
        # parse and process individual constraint string
        # filter out valid students according to string
        # check for hard and soft constraints
        constraint_list.append(c)
    return constraint_list

def gen_formula(student_constraints, course_constraints):
    formula = SATFormula()
    # convert student and course constraints into clauses
    # iterate over clauses and add them in formula
    return formula
```

```
def output(allocated_students,DB_URL):
    status = update_records(allocated_students,DB_URL)
    if status:
        post_allocation(allocated_students,DB_URL)

def post_allocation(allocated_students,DB_URL):
    notify_allocation(allocated_students)
    professor_feedback=submit_feedback(allocated_students,DB_URL)
    distribute_incentive(professor_feedback)


def process_solution(solution):
    """
    parses the solution to update the TA allocation record
    """
    allocated_students=[]
    for course in solution.courses:
        allocated_students.append(AllocatedStudents(**kwargs))
    return allocated_students

def notify_allocation(allocated_students):
    for course,students in allocated_students:
        #Send mail to professor of the course about the students selected
        for student in students:
            #Send mail to student about the course for which they are selected

def update_records(allocated_students,DB_URL):
    db_manager = database_manager.get_connection(DB_URL)
    for course,students in allocated_students:
        database_manager.update(records,course,students)

    if successful:
        return True
    return False

def submit_feedback(allocated_students,DB_URL):
    db_manager = database_manager.get_connection(DB_URL)
    professor_feedback=[]
    for course,students in allocated_students:
        db_manager.create_table(course_name,id=...,name=...)
        for student in students:
            #get feedback from professor
            professor_feedback.append((student,feedback))
            db_manager.insert(course_name,...)
    return professor_feedback

def distribute_incentive(professor_feedback):
    for student,_ in professor_feedback:
        #Distribute certificate and payment according to feedback
```