Software Design Document for Automatic Recognition of Suitable Design Pattern Project

Clara Kamal, Farida Mohamed, Hashem Mohamed, Veronia Emad
Supervised By: Dr. Taraggy Mohiy and Eng. Nada Ayman

March 9, 2020

1 Introduction

1.1 Purpose

The purpose of the software design document is to provide a full description of the architecture and the system design of our system Automatic Recognition of Suitable Design Pattern. It provides a full description about the system phases, inputs, outputs and the core algorithms used. This document is intended to developers and the designers of the project.

1.2 Scope

The system's scope can be described as an automatic selection of the suitable design pattern depending on the user requirements using a Q/A approach through a GQM-Based Tree Model. It is designed to support software engineers who need help in selecting the design patterns due to lack of knowledge or experience. It will help in saving time and cost of code refactoring in case of wrong DP selections. Moreover, the system is considered as a control point for managing the back-end composition by the system administrator. This document presents the characteristics, objectives and constraints for each of the users: software engineer and system administrator.

1.3 Overview

According to the standards for Software Design Documentation explained in "IEEE Recommended Practice for Software Design Documentation", the document is divided into 9 core sections which are:

- Introduction:
  Presents a brief description for the document
- System Overview:
  Shows the whole representation of the system's functionality and design
- System Architecture:
  Describes the complete architectural design of the system using diagrams for clarification to explain how the final design is to be achieved by connecting all these subsystems of the main system.
- Data Design:
  Includes the data structures to be used, databases and data storage units of the system.
- Component Design:
  Describe the core approach of the system
- Human Interface Design:
  Represents the UI of the system
- Requirements Matrix:
  Traces user requirements
- APPENDICES
- References

1.4 Reference Material

Since DPs are documented solutions in the form of ready-made templates, official documents and specialized books are used during the different phases of the system creation. Questions are being extracted from the definitions and keywords of each design pattern found in Design Patterns Elements of Reusable Object-Oriented Software [4] and Dive Into Design Patterns [8]. To facilitate the process of extracting the main information in each DP, a summarized format was created for the 23 DPs as shown from Table 5 to Table 25. Moreover, the scenarios used in testing was extracted from Dive Into Design Patterns [8] book.

1.5 Definitions and Acronyms

Table 1: Definitions and Acronyms

Term	Definition
Software Design Document (SDD)	Used as the primary medium for communicating software design information.
Design Entity	An element of a design that is structurally and functionally distinct from other elements.
Software Engineer (SWE)	The person responsible for the development of a software based on the principles of software engineering.
System Administrator	The person responsible for running the system.
GQM	3-layered approach that identifies goals, questions and metrics in the form of a tree. Goals are identified at the top, questions needed to reach these goals in the middle and at last the metrics that are the answers of the questions.
GUI	Graphical user interface.
UI	User Interface.
UML	Unified Modeling Language.
PUML	PlantUMLs.
MVC	Model-View-Controller design pattern.
EAV	Entity-Attribute-Value data model.
Q/A	Question/Answer.
NLP	Natural Language Processing

2 System Overview

The proposed system overview as shown in Figure 1, represents the system's users who are software engineers. The first step is the category selection. DP Category Questions are retrieved from the database in order to fill the questions model. The user will start answering the form of questions consequently and inserting his objects and classes. Next, the answers will be transmitted to the weight calculation phase to be evaluated with the aid of the GQM-based tree model. The weight of each metric (answer) will be retrieved in order to calculate the total weight for each category. Hence, the highest weight value from the three categories will be the selected category. The next step is retrieving the DPs questions for the selected category in the previous step. The user will start answer the questions and the same process of selecting the appropriate category will be repeated in order to select the suitable DP. The highest weight value will be the selected DP. The last step will be the class diagram generation. It will use the selected DP from the previous step and the objects and classes that

the user inserted at the beginning to generate the formatted PUML file to be applied to PlantUML API which will generate the class diagram.

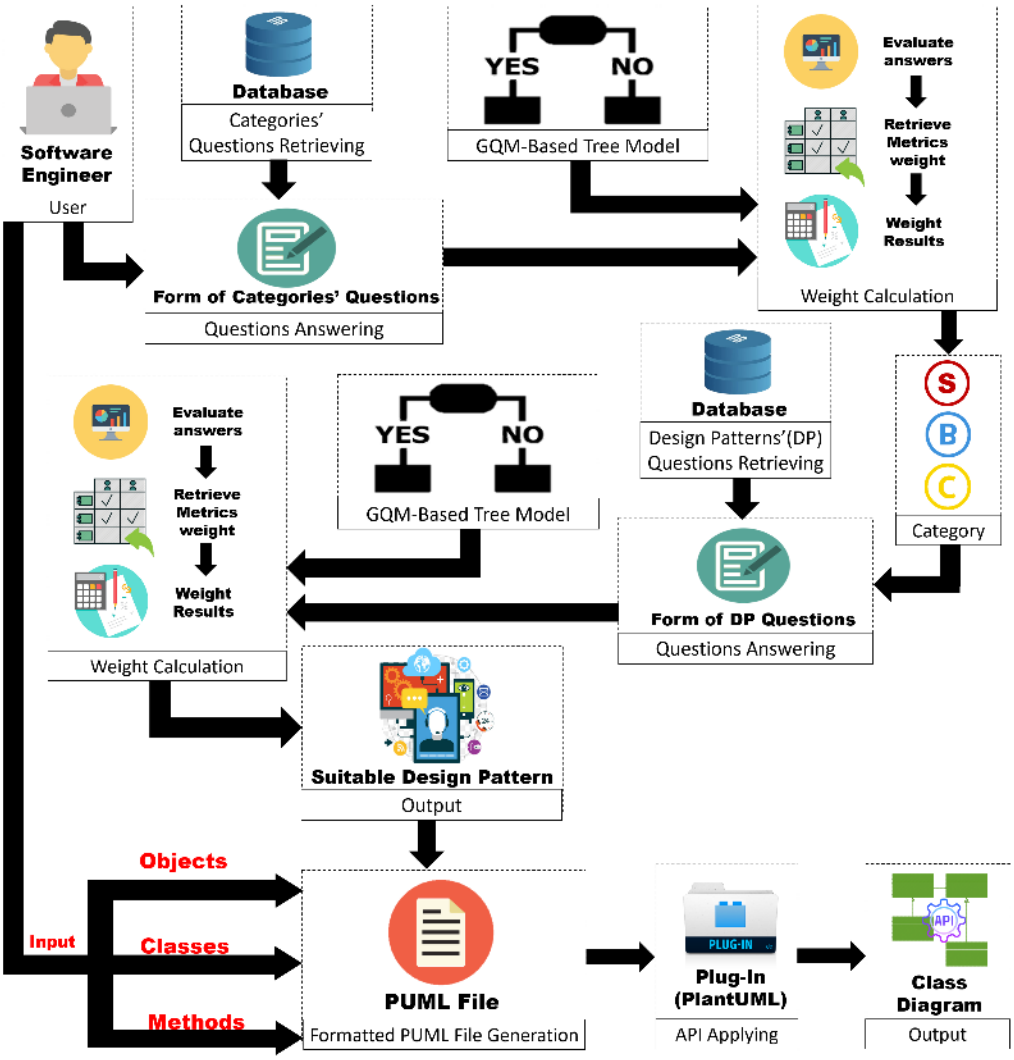


Figure 1: System Overview

3 System Architecture

3.1 Architectural Design

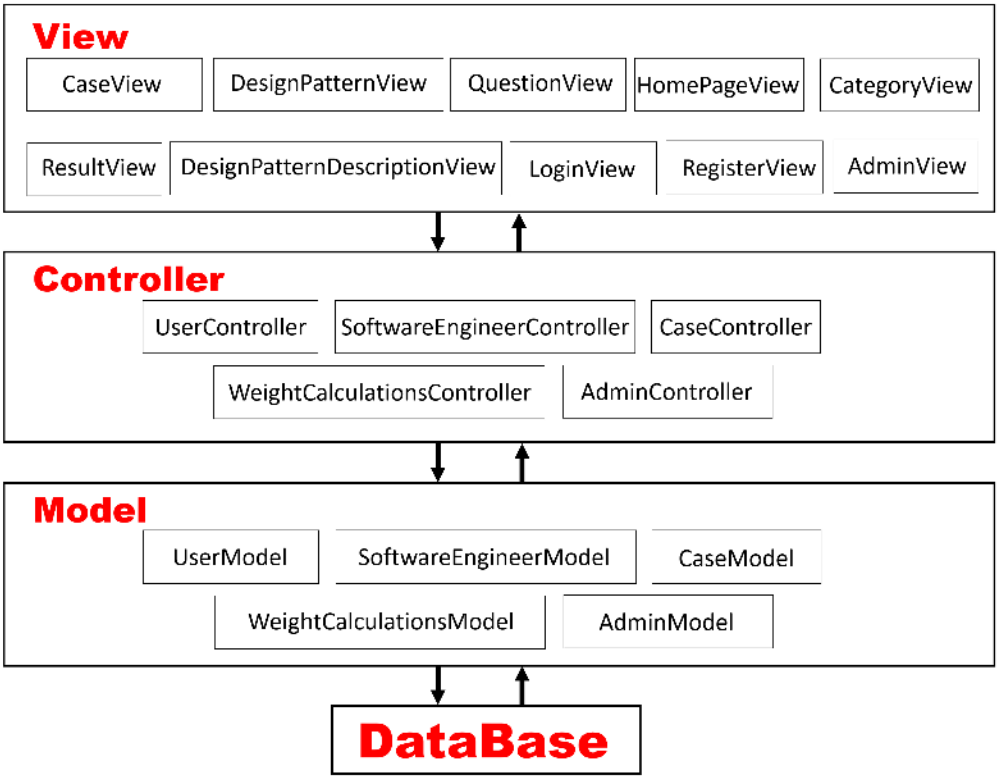


Figure 2: MVC Diagram

3.1.1 View

It is responsible for the presentation of data and representing the User Interface (UI). We have two different interfaces one is responsible for Admin operations and the other one is responsible for representing the core part including Category, Design Pattern, Question, Weight Calculation, Answer, GQM and Class Diagram. Also Category and Design Pattern both of them include "Name" and "Description". Moreover Login, Register and Result have view only in our system.

3.1.2 Controller

It is responsible for binding the view and model. The interactions and requests made within the view are taken and sent to the database to fetch data with the use of models then it forward data to the view again to be shown. Some of the controllers we are having; Admin Controller that is responsible for handling interactions made within Admin Views, User Controller that is responsible for handling interactions made by the Category, Design Pattern and Question. Also classification Controllers are responsible for core functionality and handling its interactions and finally the Notification Controller that is responsible for sending notifications to the doctor when the results are shown.

3.2 Decomposition Description

3.2.1 Class Diagram

Class name: UserType
Type: Concrete.
List of super classes: N/A.
List of sub classes.
Purpose: Class that specifies user type .
Collaboration:This class gets ggregated by UserModel class
Attributes:Integer UserTypeID, String UserType.
Operations:None.

Class name: UserModel
Type: Concrete.
List of super classes: N/A.
List of sub classes: SoftwareEngModel, AdminModel.
Purpose: Class that retrieves and store the data from the database .
Collaboration: This class aggregates class UserController,UserType
Attributes: Id,FirstName,LastName password, Email.
Operations: Register(String FirstName, String LastName , String EmailAddress,String Password),
login(String EmailAddress, String Password),
logout(), UpdateInfo(String FirstName, String LastName ,String EmailAddress,String Password) ,
ViewHomePage(), ViewResult(Integer MaxScore), PrintResult().

Class name: UserController
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: a Class that encapsulates different types of user their common attributes.
Attributes: None
Operations: Register(), login(), logout(), UpdateInfo() , ViewHomePage(), ViewResult(), PrintResult().

Class name: AdminModel
Type: Concrete.
List of super classes:UserModel.
Purpose: Class that retrieves and store the data from the database related to the users with type admin .
Collaboration: This class aggregates class AdminController, Inherits class UserModel
Attributes: None
Operations: AddAdmin(String FirstName, String LastName ,String EmailAd-

dress,String Password)
, ViewUser(Integer UserID),
DeleteUser(Integer UserID), ViewAllUsers(), ViewAllCategories()
, ViewAllDesignPatterns(), InsertCategory(String CategoryName, String Description,String Question, Integer YesWeight, Integer NoWeight),
ViewCategory(Integer CategoryID), UpdateCategory(Integer CategoryID , String CategoryName,
String Description,String Question, Integer YesWeight, Integer NoWeight), DeleteCategory(Integer CategoryID),
InsertDesignPattern(Int DesignPatternID ,Integer CategoryID, String DesignPatternName,
String Description,String Question, Integer YesWeight, Integer NoWeight) ,
ViewDesignPattern(Integer DesignPatternID),
UpdateDesignPattern(Int DesignPatternID ,Integer CategoryID, String DesignPatternName, String Description,String Question, Integer YesWeight, Integer NoWeight)
, DeleteDesignPattern(Integer DesignPatternID).

Class name: AdminController
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: To control all admin's views
Collaboration: this class aggregated by AdminView,DesignPatternView,CategoriesView.
Attributes: None
Operations: AddAdmin(), ViewUser(), DeleteUser(), ViewAllUsers(), ViewAllCategories(), ViewAllDesignPatterns(), InsertCategory(), ViewCategory(), UpdateCategory(), DeleteCategory(), InsertDesignPattern() , ViewDesignPattern(), UpdateDesignPattern() , DeleteDesignPattern().

Class name: WeightCalculationModel
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: class that retrieves and store the data from the database related to the weight calculations.
Collaboration: Aggregates WeightCalculationController,QA
Attributes: AnswerID: Integer , QuestionID: Integer , Question: String, QuestionWeight: Integer, QuestionNumber: Integer.
Operations: GenerateQuestions(QA: Question[]), NextQuestion(Integer QuestionID,String Question), PreviousQuestion(Integer QuestionID,String Question), CalculateResult(QA: Question[]);

Class name: WeightCalculationController
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: class that deals with all the operations related to weight calculations.
Collaboration: Aggregates ResultView.
Attributes: QA: Question[].
Operations: GenerateQuestions(), NextQuestion(), PreviousQuestion(), CalculateResult();

Class name: CaseModel
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: class that retrieves and store the data from the database
related to the Case need to be solved by the software engineer.
Collaboration: Aggregates CaseController.
Attributes: CaseID: Integer , Date: String, Description: String.
Operations: ViewCase(Integer CaseID,Integer UserID,String Date,String Description)
, DeleteAllCases().

Class name: CaseController
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: To control case view.
Collaboration: Gets Aggregated by CaseModel and CaseView.
Attributes: None Operations: ViewCase() , DeleteAllCases().

Class name: CaseView
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: to represent the case view .
Collaboration: Aggregates CaseController .
Attributes: None .
Operations: ViewCase() , DeleteAllCases().

Class name: QA
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: Includes all components needed to represent an object composed of question ,answer and weight .
Collaboration: Aggregated by WeightCalculationModel .
Attributes: Integer AnswerID , Integer QuestionID,
Integer WeightID, String Question.
Operations:None .

Class name: SoftwareEngModel
Type: Concrete.
List of super classes: UserModel.
List of sub classes: N/A..
Purpose: class that retrieves and store the data from the database related to the software engineer.
Collaboration:Inherits UserModel , aggregates SoftwareEngController .
Attributes: None.
Operations: AnswerQuestions(IntegerQuestionID)
, ViewHistory()
, DeleteHistory(Integer HistoryID),DeleteAllHistory(),.
ViewDpDescription();

Class name: SoftwareEngModelController
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: To control software engineer views.
Collaboration:Assists CaseModel.
Attributes: None Operations: AnswerQuestions()
, ViewHistory()
, DeleteHistory(),DeleteAllHistory(),.
ViewDpDescription();

Class name: AdminView
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: to represent the Admin view .
Collaboration: Aggregates AdminController .
Attributes: None .
Operations: AddUser(), ViewUser(),
ViewAllUsers(), DeleteUser().

Class name: DesignPatternView
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: to represent the Design Pattern view .
Collaboration: Aggregates AdminController .
Attributes: None .
Operations: InsertDesignPattern(), ViewDesignPattern(),
UpdateDesignPattern(), DeleteDesignPattern()
, ViewAllDesignPatterns() .

Class name: CategoryView
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: to represent the category view.
Collaboration: Aggregates AdminController .
Attributes: None .
Operations: InsertCategory(), ViewCategory(),
UpdateCategory(), DeleteCategory()
, ViewAllCategories() .

Class name: DesignPattern
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: creating design pattern and categories objects with their all possible attributes.
Collaboration: Aggregates AdminController .
Attributes: Integer CategoryID, String CategoryName,
String CategoryDescription, 0
Integer DesignPatternID, String DesignPatternName,
String DesignPatternDescription.
Operations: none.

Class name: RegisterView
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: to represent the Register form view.
Collaboration: Aggregates SoftwareEngController .
Attributes: None .
Operations: Register() .

Class name: HomePageView
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: to represent the Homepage view.
Collaboration: Aggregates UserController .
Attributes: None .
Operations: ViewProfile(), ViewDescriptions(),
ViewUpdateInfo(), QuestionsView()

Class name: DesignPatternDescriptionView
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: to represent the DesignPattern and categories descriptionn view.
Collaboration: Aggregates UserController .
Attributes: None .
Operations: ViewDpDescriptions().

Class name: LoginView
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: to represent the login view.
Collaboration: Aggregates UserController .
Attributes: None .
Operations: Loginview().

Class name: ResultView
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: to represent the Result of the suitable category and design pattern view.
Collaboration: Aggregates UserController .
Attributes: None .
Operations: ViewResult(),PrintResult().

Class name: QuestionView
Type: Concrete.
List of super classes: N/A.
List of sub classes: N/A..
Purpose: to represent the Question's form view.
Collaboration: Aggregates UserController,QA and assists WeightCalculationModel .
Attributes: QA: Question[] .
Operations: AnswerQuestions(), GenerateQuestions(),
NextQuestion(), PreviousQuestion.

Class name: PrintResult.
Type: Interface.
List of super classes: None.
List of sub classes: None.
Purpose: To allow printing results with different printing strategies.
Collaboration: Class PrintResultPDFbyDate and PrintResultPDFbyName implements this class.
Attributes: None.
Operations: PrintResult().

Class name: PrintResultPDFbyDate.
Type: concrete.
List of super classes: None.
List of sub classes: None.
Purpose: To allow printing results by date.
Collaboration: This class implements class PrintResult.
Attributes: None.
Operations:PrintResult().

Class name: PrintResultPDFbyName.
Type: concrete.
List of super classes: None.
List of sub classes: None.
Purpose: To allow printing results by name.
Collaboration: This class implements class PrintResult.
Attributes: None.
Operations:PrintResult().

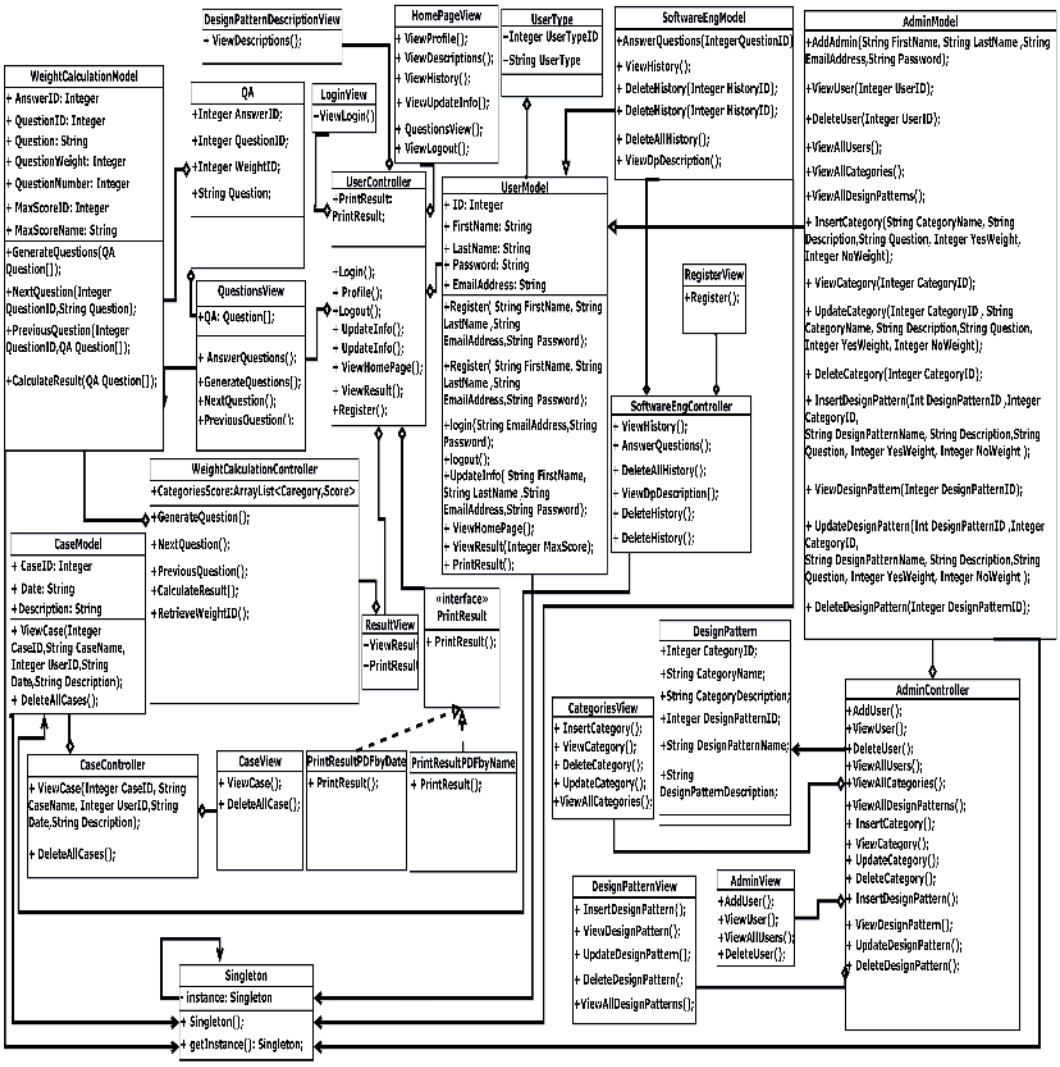


Figure 3: Class Diagram

3.2.2 State Diagram

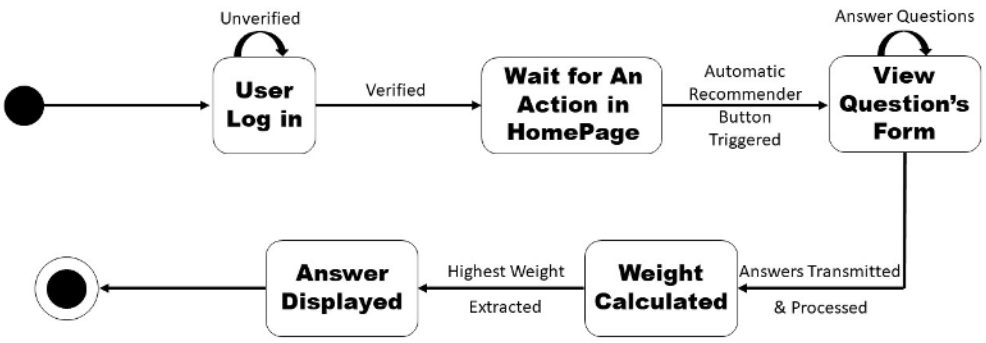
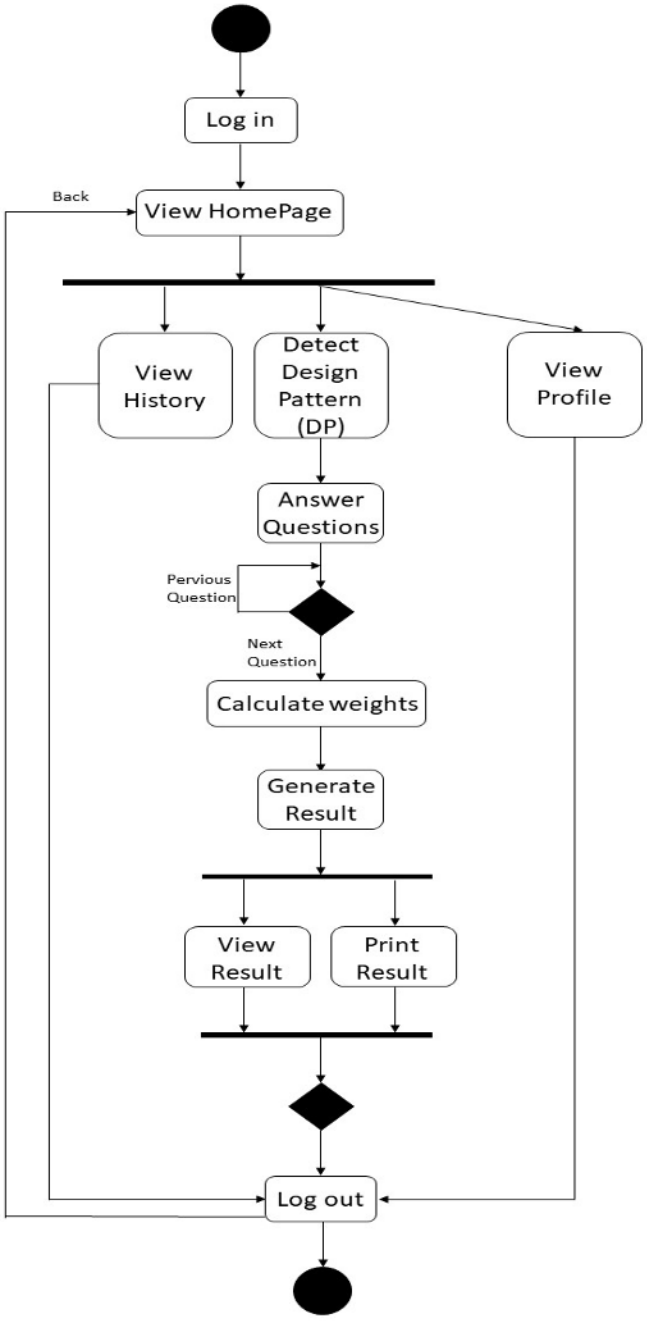


Figure 4: State Diagram

3.2.3 Activity Diagram



3.2.4 Block Diagram

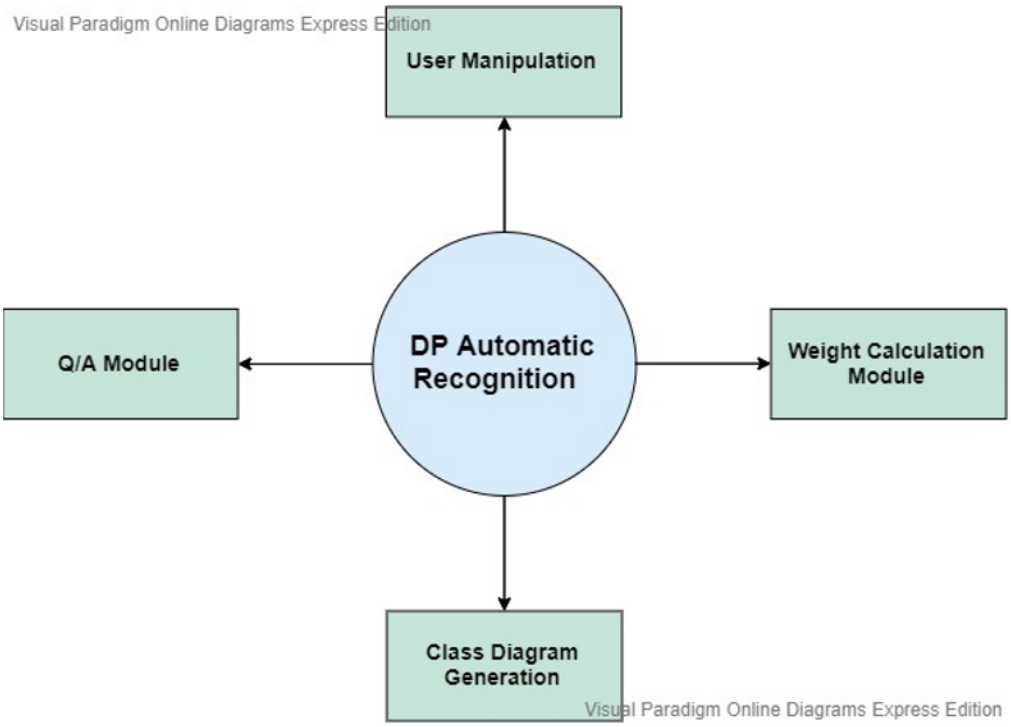


Figure 6: Block Diagram

3.2.5 Process Diagram

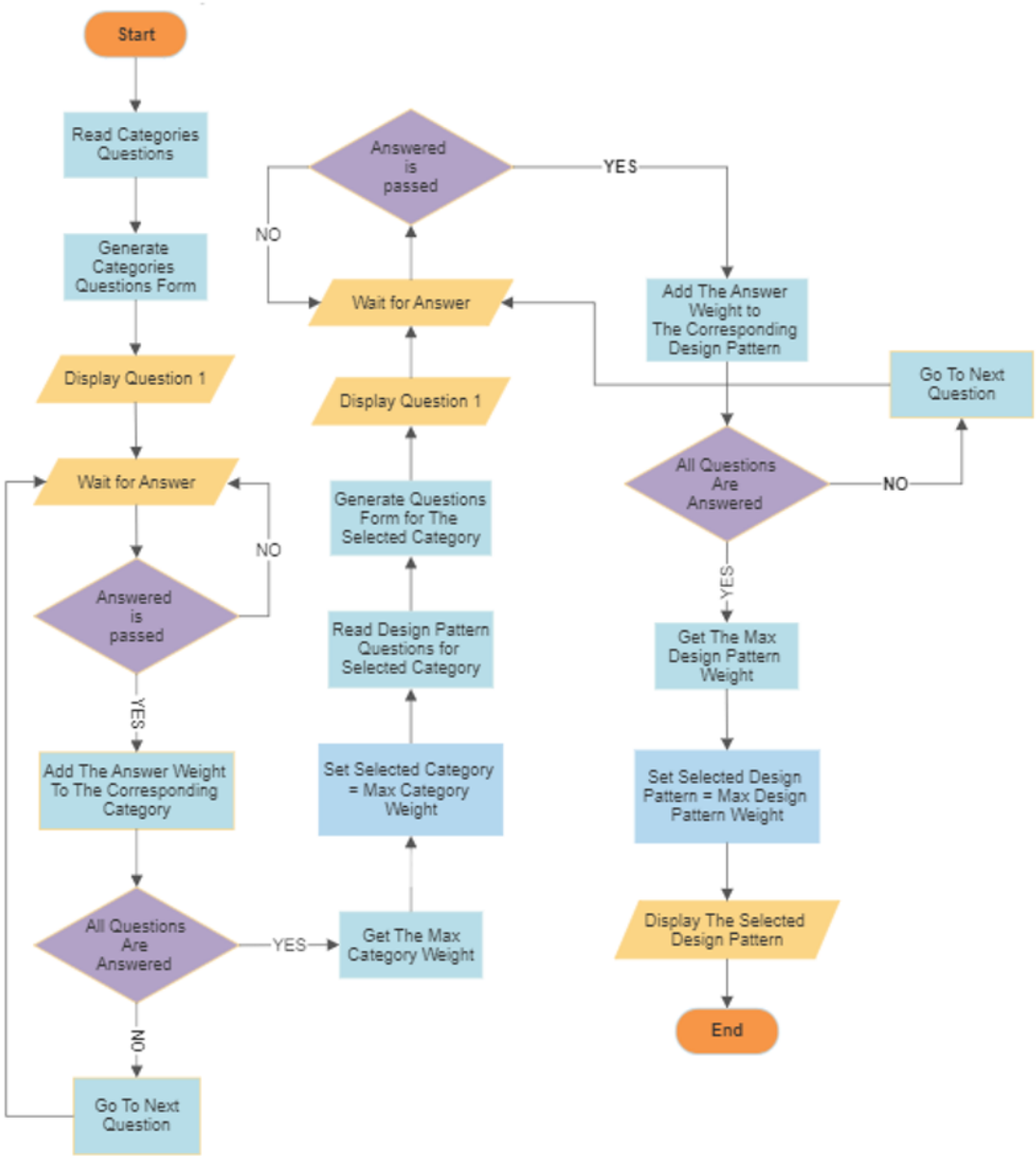


Figure 7: Process Diagram

3.2.6 Sequence Diagram

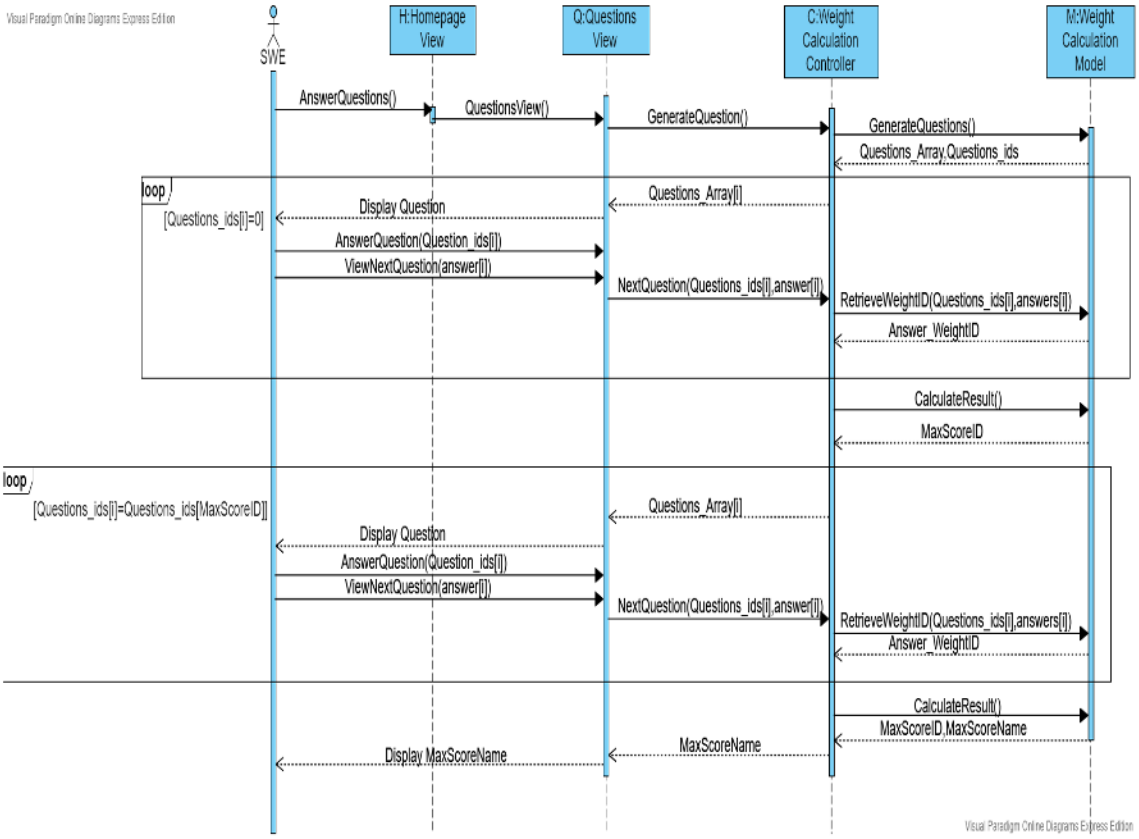


Figure 8: Sequence Diagram

3.3 Design Rationale

As mentioned previously through Figure 2, the system's architecture is based on MVC. The system is structured into three logical components that interact with each other.It helps in separating data presentation and user interaction from the system data and functionality. Moreover, it provides flexibility for future modifications and code reusability.

Regarding our current problem that needs to be solved, which is the suitable selection of design patterns for specific design problems, we found different approaches to solve it.

- **GQM approach**:
  3-layered approach that identifies goals, questions and metrics in the form of a tree. Goals should be identified at the top then the questions that are needed to reach these goals and at last the metrics that are the answers of the questions. Weights are assigned to each of these answers and the design pattern with the highest weight will be the selected design pattern. This approach is represented in the form of Yes/No questions.

- **NLP Approach**:
  Takes the user's problem in the form of English description. It starts processing the description by removing unnecessary words, highlight adjectives and verbs, matching the results with the corresponding keywords of each design pattern and finally recognize the suitable design pattern. This approach can be represented in the of essay questions that specifies the type of requirements needed to be entered. Another representation can be by giving the user the flexibility to type his problem from his point of view.

As a result of analyzing both approaches, we selected the GQM approach to be our core approach. We found that GQM provides a good differentiation approach using weights and it doesn't depend on the sequence of answers but the final weights. Moreover, by applying Q/A approach, it will help in extracting more specific and effective requirements from the user through the specific questions and answers provided to the user.

4 Data Design

4.1 Data Description

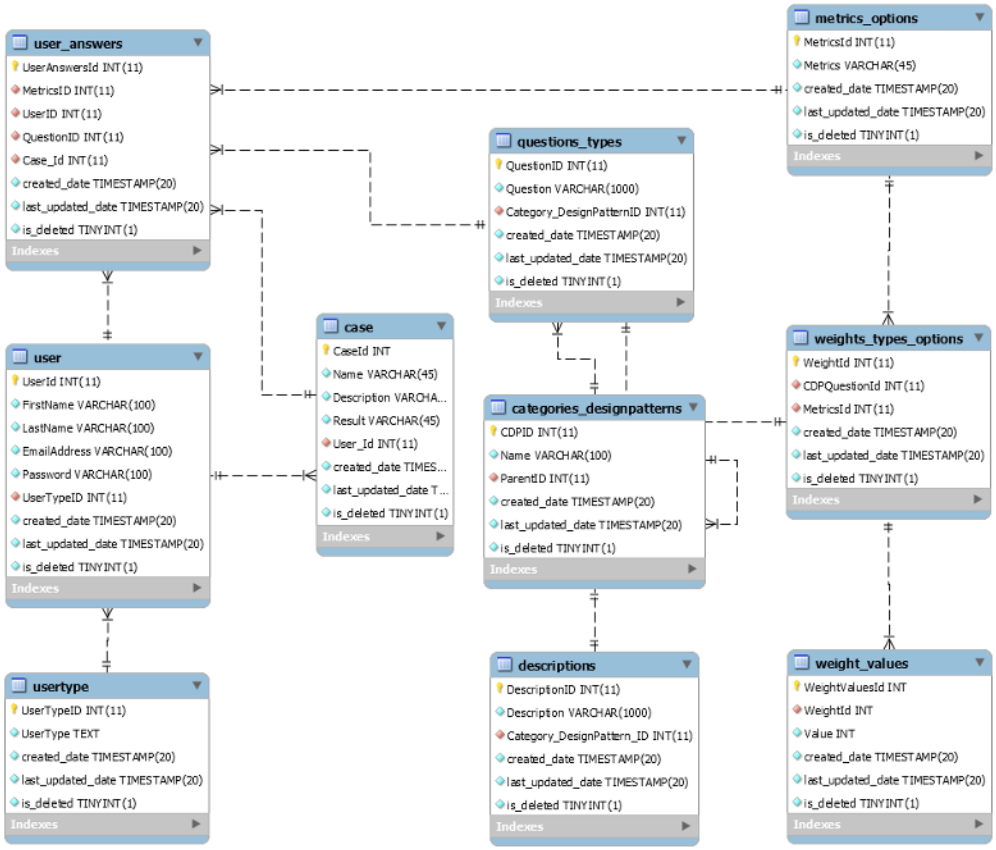


Figure 9: System ERD

The "User" table contains the personal information of any user and the UserType attribute (Foreign Key) to differentiate between the users as the system has two usertypes: Admin and Software Engineer, which can be increased in the future that's why we need the "Usertype" table which contains the types of the users as each type has its own different privileges and permissions.

The "categories_designpatterns" table contains the name of the category or the design pattern and its parentID, as it relies on the Parent/Child Relationship table design concepts which means that it's hierarchical, because each category is the parent of multiple design patterns. So this way we prevent duplicating multiple tables with same attributes. Each Category/Design Pattern has a description which is stored in the "descriptions" table.

The system is also based on the Entity Attribute Model concept which basically consists of entities table where the number of their attributes that describe them are more likely to be expanded or increased in the future, but the number of attributes are the options that vary with each different type of entity; then the values table which stores the value that is linked to an attribute. So this concept applies on our system in the case of the questions, their possible answers and these answers' weights (values).

The "questions_types" table include the Category_DesignPatternID passed as a Foreign Key from the "categories_designpatterns" table, which stores each Category or Design Pattern's corresponding questions.

The "metrics_options" table contains all the possible answers that the questions could have.

The "weights_types_options" table contains the question ID and the Metric ID in which each question can have a couple of metrics (answers).

Each user has his/her answers saved in a table called "user_answers", which stores each questions and its corresponding answer that is answered before by the user, in addition to the case ID which is a Foreign Key from the "case" table.

The "case" table simply store the name of each case or scenario that the user answered the questions based on it. It's used for later reference or as a history for the user in case of facing a similar problem, he/she won't need to answer all the questions again.

At last, Each table has the three constant attributes of the log files which are: created_date, last_updated_date and is_deleted to prevent any record from permanent loss and erasing the historical data.

4.2 Data Dictionary

4.2.1 Security

- User's sensitive information such as passwords should never be displayed on screen. It should be encrypted using special characters.
- Each record in the database should be encrypted so if any data changed, the changed row can be detected easily.
- The system must verify each email entered through the registration process using an email matching function.

4.2.2 Reliability

Reliability is achieved by applying singleton design pattern that ensures that only one instance the database class is created and that object is shared across all the clients.

4.2.3 Maintainability

Maintainability of the system will be achieved by using MVC design pattern that divides the system into three modules to separate the data handling in each module. The system should provide a straight-forward friendly GUI that allows the system administrator to add, update and delete any of the accessible database information easily such as the questions in the tree structure behind the design pattern selection process. Moreover, the system is able to serve multiple user types by applying EAV in the user classes.

4.2.4 Extensibility

The system is easy to be extended and updated with the minimum effect by applying EAV data model. The system administrator can add new questions and update or delete any of the existing questions without affecting the tree structure behind the design pattern selection process.

5 Component Design

5.1 System Approach

Our proposed system aims to provide an automatic selection of the suitable design pattern through a GQM-based tree model. The proposed tree model includes a sequence of questions that the software developer needs to answer. These questions are designed upon the definitions of the three categories: Creational, Structural and Behavioral and the 23 design patterns [4]. Based on the answers, the flow of the tree model will lead to the corresponding design pattern. The developer's requirements are extracted from the questions' answers and the weights of the answers leads to the suitable design pattern.

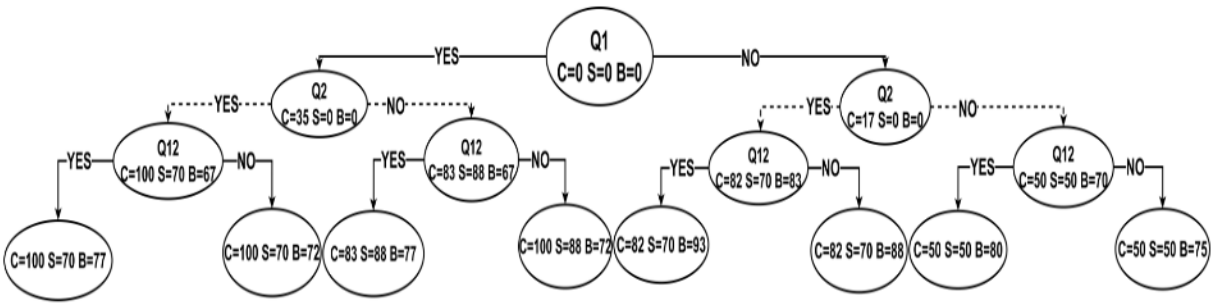


Figure 10: GQM-Based Tree Model

A sample of the GQM based tree model is represented in Figure 10. It shows the first level, last level and the dots which represent the intermediate levels. It is a combination between GQM tree and decision tree. This tree consists of 12 levels of design patterns categories' questions. Each question has two possible answers which are yes and no and different yes and no weights. The weight is calculated depending on the sum of answers' weight of each question.

Unlike other approaches, this approach starts by classifying the category of the DP using different sets of questions that define these categories. After the system decides which category is the most relevant to the user's problem scenario, it will go through the design patterns' questions of this category only. This will reduce the probabilities of the potential suitable pattern. Moreover, it will reduce the users' confusion which is resulted from asking them many questions about irrelevant patterns which will eventually keep them away from the desired result and will take more time. Therefore, this approach help in automating this process even more accurate and faster.

The weights of questions' answers are shown in Table 2. The first four questions concern "the Creational" category, Questions 5 to 7 concerns "the Structural" category and from 8 to 12 affects "the Behavioral" category. Starting from Question 13 to 24 concerns the Design Patterns of "the Creational" Category. "The Abstract Factory" (AF) DP will be affected by the Question 13 and 14, while Question 16 and 17 concerns "the Factory Method" (FM) DP, in addition to Question 15 that affects both of them. The three consecutive questions starting from Question 18 to Question 20 concerns "the Builder" (BD) DP. Then Question 21, 22 affects "the Prototype" (P) DP; and the last two questions concerns "the Singleton" (ST) DP category. The 'No' answer always weighs 50% of the 'Yes' answer weight.

Table 2: Question/Answer Weight Model

Question ID	**Question**	**Yes Weight**	**No Weight**	**Affected Category / DP**
Q1	Is this class more concerned about the way the objects are created?	35	17	C
Q2	Taking a "maze game" as an example, does this problem ignore details of what can be in a maze and whether a maze game has a single or multiple players. Instead, it just focus on how mazes get created in the first place?	30	15	C
Q3	Do you need to configure a system with objects that vary widely in structure and functionality?	20	10	C
Q4	Does creating an object using the new operator increase the complexity of your code?	15	8	C
Q5	Will this class be more concerned about how the object is composed of group of objects from different classes, to form larger structures?	40	20	S
Q6	Would you like to combine the objects, interfaces or implementations of multiple classes together to obtain new functionality?	35	17	S
Q7	Will this class have any relationships between it and any other classes in the same problem?	25	13	S
Q8	Will this classes be Concerned with interaction, responsibility and communication between objects?	30	15	B
Q9	Are the steps of a task are divided among different objects?	25	12	B
Q10	Does your problems scenario needs a flow of processes?	20	10	B
Q11	Does the object behaviour should be represented dynamically?	15	8	B
Q12	Would your class control whether the objects to be dependent or independent?	10	5	B

Q13	Does this case deals with a group of related items?	45	23	AF
Q14	Is this case more concerned about creating the object itself rather than its type?	30	15	AF
Q15	types of this case's objects could vary or increase in the future?	25	12	AF & FM
Q16	Does all case's objects are based on a common base but differ in their internal components? (hint in pizza store pizza is the commen base but have different internal components as chicken or peppproni)	45	23	FM
Q17	Would this case depend on reusing existing objects instead of rebuilding them each time?	30	15	FM
Q18	Will this case produce different types and representations of objects using the same method of construction?	45	23	BD
Q19	Does this case contains complex objects that have certain configurations which makes constructing them step-by-step necessary?	35	17	BD
Q20	Does constructing the objects step-by-step doesn't need all of the steps to be called?	20	10	BD
Q21	Do you need to carry over all of the field values of the old object into the new one?(Cloning)?	55	28	P
Q22	Would cloning existing objects that match in some configuration reduce the number of subclasses?	45	22	P
Q23	Do many Classes in this case depend on one global variable that can't be overwritten and would be repeated in each one of them?	60	30	ST
Q24	Does the case have some shared resources? (for example, a database or a file.)	40	20	ST

5.2 Weights Assigning

The algorithm shown below explains the steps of how the weights were evaluated based on the Recommendation System for Design Patterns in Software Development Conference Paper [7].

1. Extract the questions from the definitions and the most common problems that each category/DP solves.
2. Set two answers to each question which are 'Yes' or 'No' only.
3. Set the total weight of the 'Yes' answers to 100 and the 'No' answers to 50, regarding each category's/DP's questions.
4. Assign the 'Yes' weights of each question depending on the importance of the problem that the question describes, and the 'No' weights which will be the half of its corresponding 'Yes' value. (Example: Yes = 40, No = 20)
5. Calculate the total sum of each category's/DP's weight.

5.3 Implementation Approach

Regarding the implementation approach, the pseudo code in Algorithm 1 retrieves firstly all the categories' questions from the database, which is a centralized database in order to improve data preservation and avoid the risk of any data loss. Then it stores them in an array called QuestionsArray. It also stores both yes and no answers weight in two different arrays called QuestionYesWeight and QuestionNoWeight. Then it displays all the questions stored in the array. The user answers each question with either yes or no answers. Depending on the user's answer the weight of this answer will be stored in its category score. The highest category score will be the suitable category that the user should use. After that the category name will be displayed for the user. The same process is applied on the design pattern to select the final result of the suitable design pattern.

Algorithm 1: Testing Approach

```
input: UserAnswer;
QuestionsArray[NumberOfQuestions]; CategoryScoreArray[3];
  //Array's indices are Creational, Structural and Behavioral.
for i = 0 to NumberOfQuestions do
    query = "Select * from Questions where ID = i";
    Question = query.executeQuery;
     QuestionsArray[i]=Question;
end
for i=0 to NumberOfQuestions do
    Print QuestionsArray[i];
    if UserAnswer is"Yes" then
        if QuestionsArray[i].category = "Creational" then
            CategoryScoreArray[1] += QuestionsArray[i].YesWeight
        end
        //This if condition also applies for the 2 other categories.
    end
    if UserAnswer is"No" then
        if QuestionsArray[i].category = "Creational" then
            CategoryScoreArray[1] += QuestionsArray[i].NoWeight
        end
        //This if condition also applies for the 2 other categories.
    end
end
//find maximum score from the 3 categories' scores.
  max(CategoryScoreArray[3]);
//Display the suitable category based on the highest category score.
```

Result: the suitable category name

5.4 Experiments and Results

The GQM base tree model was tested on 10 case studies to check if it could recognize the suitable categories (Creational, Structural,Behavioral).
These case studies were retrieved from two books [4] [8]. Four out of the ten case studies are illustrated below.

Case Study 1: The customer calls the customer service's number to speak with a customer services agent. The customer service agent helps in providing an interface to the billing department, the shipping department and the order fulfillment department. In order to be able to finish all the steps required to complete the customer's order. Implementing this system should start with creating an unified interface for the element or subsystem. Then there should be a class that is called 'wrapper' class that is responsible to encapsulate the subsystem. The wrapper class detects the complexity and associations of the elements and assigns them to the appropriate methods.This case study was solved by going through the GQM-based tree model shown in Figure 11.

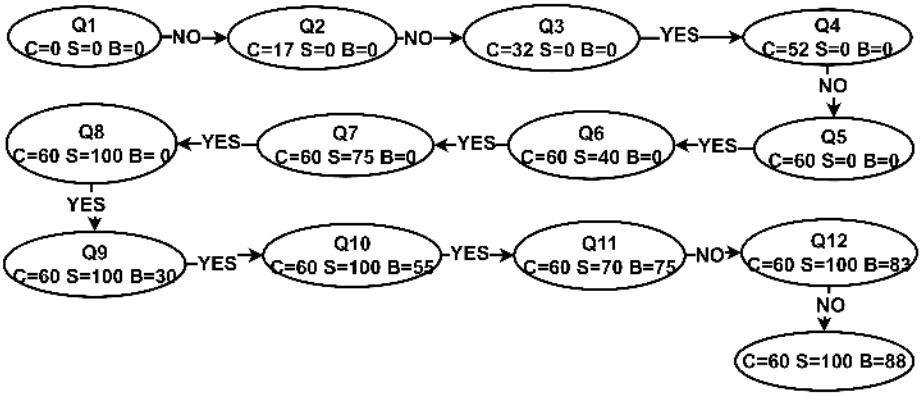


Figure 11: Case Study 1 Tree Optimum Path

The reasons behind each answer shown in Figure 11 is mentioned below:
1- No, The customer services class is more concerned about combining objects not creating them.
2- No, This system doesn't not ignore the details because in this case the customer service class is mainly concerned about the details and the number of departments the customer service has.
3- Yes , Because each department has it's own structure and functionalities.
4- No, This system is not really concerned about neither code complexity nor creating objects.
5- Yes , This class is about how the object is composed of group of objects from different classes such as the placing an order that many classes interfere in starting from the warehouse till the shipping department.
6- Yes, as it will make this system easier to implement.
9- Yes, Almost all the tasks in this system are divided among different objects.
10- Yes, Because each order has a certain sequence that must be followed.

Case Study 2: The United States Constitution determines the factors by which a president is elected, by limiting the term of office, and clarifies the order of succession. There can be at most one president at any given time. Despite of the personal identity of the current president, the title, "The President of the United States" is a global point of access that describes the person in the office. The main concern of this system is to create an object which is the president of the united states regardless any other detail. This case study was solved by going through the GQM-based tree model shown in Figure 12.

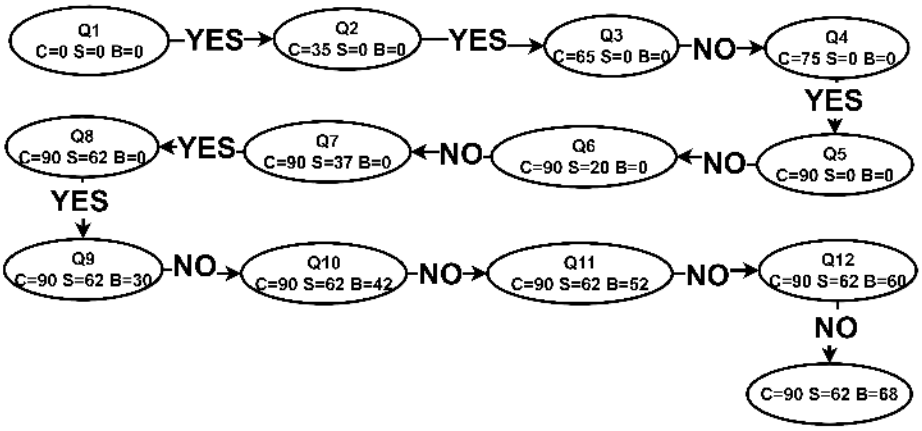


Figure 12: Case Study 2 Tree Optimum Path

The reasons behind each answer shown in Figure 12 is mentioned below:
1- Yes, the United States Constitution class is more concerned about creating only one instance of the President object.
2- Yes, this system doesn't focus on any details as mentioned in the case's scenario: " The main concern of this system is to create an object which is the president of the united states regardless any other detail".
3- No , having objects that vary extensively is not needed because we only need the President object for one purpose and functionality.
4- Yes, the system doesn't need to be provided with several President objects so it creates complexity, as mentioned in the case study: "The President of the United States is a global point of access".
7- Yes, All the classes will have relationships between them.
8- Yes , because the United States Constitution class must communicate and interact with many objects such as the president and the rest of the office employees.
9- No, they're not divided among different objects because it depends only on the President.
10- No, it consists of one step only which is determining the factors of the elected President.
11- No, the President has fixed responsibilities and functionalities.

Case Study 3: Designing an editor for creating documents that includes text and graphics. There are some applications that should be done on text as spelling, searching and replacing. There are different types of elements inside the document. It can be character, line, page, and shapes. This system is composed of one main class which is the document itself and other sub-classes that combine together form this document. Which means that we can have a class called Picture that has several components such as pixels and color and size. Another class which is Shape that has another components such as length, width, volume and color.These two classes combine together so the document can include both pictures and shapes. The This case study was solved by going through the GQM-based tree model shown in Figure 13.

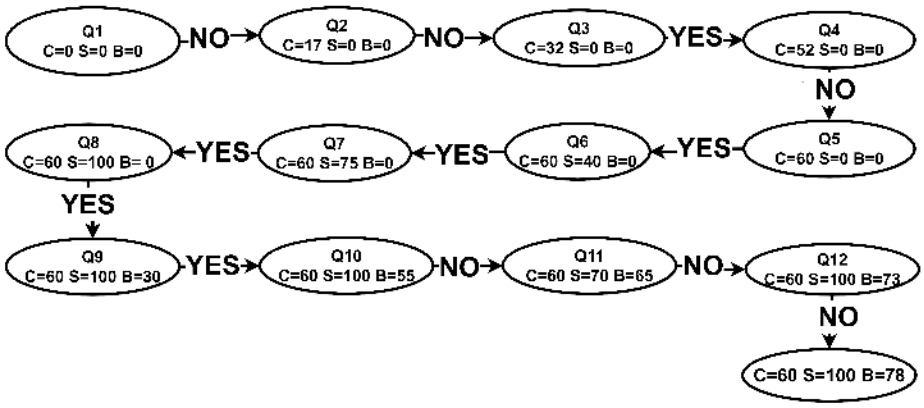


Figure 13: Case Study 3 Tree Optimum Path

The reasons behind each answer shown in Figure 13 is mentioned below:
1- No, the text editor class is more focused on the combination of the different objects not on their creation.
2- No, because its main concern is the details included in the text editor not the creation of the text editor itself.
3- Yes, because each document will contain the same objects but differ in structure.
5- Yes , because it depends on many objects from different classes together to create the whole document.
6- Yes, because the text editor is mainly based on the combination of objects and elements like the characters, lines, pages, and shapes. As mentioned also in the scenario: "This system is composed of one main class which is the document itself and other sub-classes that combine together form this document".
7- Yes, the system's classes will have relationships between other classes like the characters, pictures and shapes.
8- Yes , the document class will have to communicate and interact with all the objects needed to be able to reach its large structure.
9- Yes, as the user needs to do several steps consecutively to reach the desired document at the end.

Case Study 4: A pizza store wants to add more than one type of pizza to its menu. Because all of the competitors have added a several of trendy pizzas to their menus: The Ranch Pizza and the Veggie Pizza. Clearly, the pizza store wants to compete with the competitors, so it added these pizzas to its menu. On the other hand the pizza shop isn't selling many Greek Pizzas lately, so it decided to lift that off the menu. The system should be flexible to add more pizza types in the future . In order to obtain this flexibility the system should have one main class which is Pizza Factory. This includes the whole process of making a pizza starting from the dough until it is ready to be delivered. The other classes of this system are the types of pizzas that have different components. Adding more type of pizza means adding one more class only .This won't affect any other classes and won't also affect the design of the system. This case study was solved by going through the GQM-based tree model shown in Figure 14.

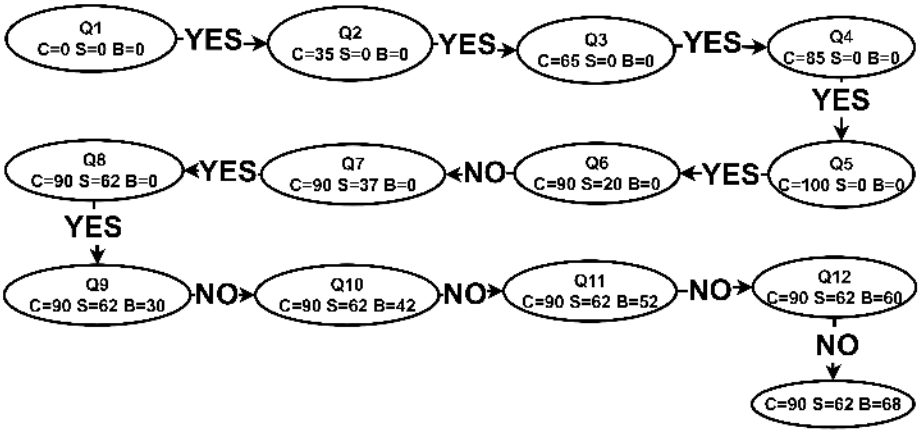


Figure 14: Case Study 4 Tree Optimum Path

The reasons behind each answer shown in Figure 14 is mentioned below:
1- No,the Pizza factory class (main class) is more concerned about combining the ingredients of the pizza to form a new pizza.
2- No, this systems is concerned about the details of each pizza because they all differ in shape, size , ingredients and taste not how the pizza itself is created.
3- Yes , because the formation of each pizza is different compared to another pizza type.
4- No, this system is mainly concerned about the adaptability of combing the classes and objects together rather than code complexity.
5- Yes, each class my have different components and objects but they all aim to form a larger formation which is the pizza itself.
8- Yes, communication will help classes and objects talk together so they can interact easily.
9- Yes, all the pizza tasks are divided among different objects, combining to-

gether to form the pizza itself.
10- Yes, There are several steps that should be followed in sequence in order to form the pizza as an example the Tomato Sauce should be added before the Mozzarella Cheese.

Regarding the case studies that were tested on the Creational's five Design Patterns, five cases were tested, one case per DP. The questions recognized the suitable Design Pattern successfully after answering the 14 Questions. These case studies were retrieved from the same two books as the Categories' [4] [8]. All of the cases are illustrated below.

Case Study 1: In travel site, we can book train ticket as well bus tickets and flight ticket. In this case user can give his travel type as 'bus', 'train' or 'flight'. Here we have an abstract class 'AnyTravel' with a static member function 'GetObject' which depending on user's travel type, will create and return object of 'BusTravel' or ' TrainTravel'. 'BusTravel' or ' TrainTravel' have common functions like passenger name, Origin, destinationparameters. The reasons behind the Factory Method answer is mentioned below:
1- Does this case deals with a group of related items? yes.
2- Is this case more concerned about creating the object itself rather than its type? no.
3- types of this case's objects could vary or increase in the future? yes.
4-Does all case's objects are based on a common base but differ in their internal components? (hint in pizza store pizza is the commen base but have different internal components as chicken or peppproni) yes.
5-Would this case depend on reusing existing objects instead of rebuilding them each time? yes.

Case Study 2: We have considered a business case of fast-food restaurant where a typical meal could be a burger and a cold drink. Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or pepsi and will be packed in a bottle. The reasons behind the Builder answer is explained below:
1- Does this case deals with a group of related items? yes .
2- Is this case more concerned about creating the object itself rather than its type? no .
3- types of this case's objects could vary or increase in the future? yes.
6- Will this case produce different types and representations of objects using the same method of construction? yes.
7- Does this case contains complex objects that have certain configurations which makes constructing them step-by-step necessary ? yes.
8- Does constructing the objects step-by-step doesn't need all of the steps to be called? yes .

Case Study 3: Let's think about the customer who has multiple Windows-based (Desktop) applications currently in use for different financial purposes. Now, they want us to provide a solution to add the printing feature in all the applications they have. One thing which is clearly stated is that they want printing functionality to be added to all their applications, which means, we need to create a printing library or class that can be shared or accessed from multiple applications. The reasons behind the Singleton answer is discussed below:
6- Will this case produce different types and representations of objects using the same method of construction? no.
8- Does constructing the objects step-by-step doesn't need all of the steps to be called? no .
11- Do many Classes in this case depend on one global variable that can't be overwritten and would be repeated in each one of them? yes.
12- Does the case have some shared resources? (for example, a database or a file.) yes.

Case Study 4: imagine we have two authors, Edgar Allan Poe and Charles Darwin, both looking to publish their (arguably) most famous works, The Raven and On the Origin of Species, respectively. Since one is a poet and the other a scientist, the types of work they've created are quite different (poem and research paper). Moreover, given those disparate types of writings, it's unlikely that they'll both be using the same publisher. As it happens, The Raven was first published in 1845 by a periodical called The American Review, while On the Origin of Species was published in 1859 by John Murray, an eclectic English publishing firm. The reasons behind the Abstract Factory answer is mentioned below:
1- Does this case deals with a group of related items? yes.
2- Is this case more concerned about creating the object itself rather than its type? yes.
3- types of this case's objects could vary or increase in the future? yes.
4-Does all case's objects are based on a common base but differ in their internal components? (hint in pizza store pizza is the common base but have different internal components as chicken or pepperoni) no.
5-Would this case depend on reusing existing objects instead of rebuilding them each time? no.
7- Does this case contains complex objects that have certain configurations which makes constructing them step-by-step necessary ? no .

Case Study 5: Consider the case of Photoshop. A graphics designer add an image to the canvas. Then, he adds a border to it. Then, he gives it a bevel effect. Finally, he sets its transparency to 5o%. Now, he wants to apply the same design to another 20 images. The reasons behind the Prototype answer is explained below:
4-Does all case's objects are based on a common base but differ in their internal components? (hint in pizza store pizza is the commen base but have different internal components as chicken or peppproni) yes.
6- Will this case produce diffrent types and representations of objects using the same method of construction? no.
7- Does this case contains complex objects that have certain configurations which makes constructing them step-by-step necessary ? yes.
8- Does constructing the objects step-by-step doesn't need all of the steps to be called? yes.
9- Do you need to carry over all of the field values of the old object into the new one?(Cloning)? yes.
10- Would cloning existing objects that match in some configuration reduce the number of subclasses? yes.

5.5 Statistics

To evaluate the efficiency of the first phase in our approach, we gathered all the experiments' results into Table 3 which represents the three results that were obtained from the ten experiments applied on the system. The true positive percentage denotes the number of times the system had selected the suitable DP category. The false positive percentage indicates the number of times the system had selected the selection of a wrong DP category that doesn't suit the given problem scenario. In the structural category's false positive percentage, the system detected two possible categories which were structural and behavioral, which means it wasn't able to select the suitable DP category for this problem scenario. The results that were obtained by the ten case studies that were tested by our system were 80% precision, 100% recall and 80% accuracy.

Table 3: Proposed Approach Categories Results

Category	**True Positive Percentage**	**False Positive Percentage**
Creational Category	75%	25%
Structural Category	66%	34%
Behavioral Category	100%	0%

A comparison with the similar systems is shown in Table 4. Outperforming of the proposed approach is due to starts with the DP categories' questions, and thus it is more effective and can be applied in more scenarios.

Table 4: Comparison with Similar Systems

Approach	**Precision**	**Specialization**
NLP [6]	93%	Class diagrams generation to case studies from Information Systems and Object-Oriented Analysis books.
NLP [3]	87%	Use Cases generation from Web Company online dataset.
GQM [7]	50%	3 DPs recognition on cases from Gof book [4].
LRNN [2]	99.6%	2 DPs recognition on collected dataset.
NLP [5]	65.5%	14 DPs recognition on collected dataset.
NLP [1]	72%	23 DPs recognition on collected dataset.
Ours	80%	3 DP Category recognition on cases from Gof book [4].

6 Human Interface Design

6.1 Overview of User Interface

Our system is a desktop application with a simple responsive UI. Two types of users can be able to log in to the system: the software engineers and system administrators. The SWE's purpose is to get the suitable DP for his requirements and the system administrator main purpose is to manage the system's data and structure. Different screens are provided to each user type.

6.2 Screen Images

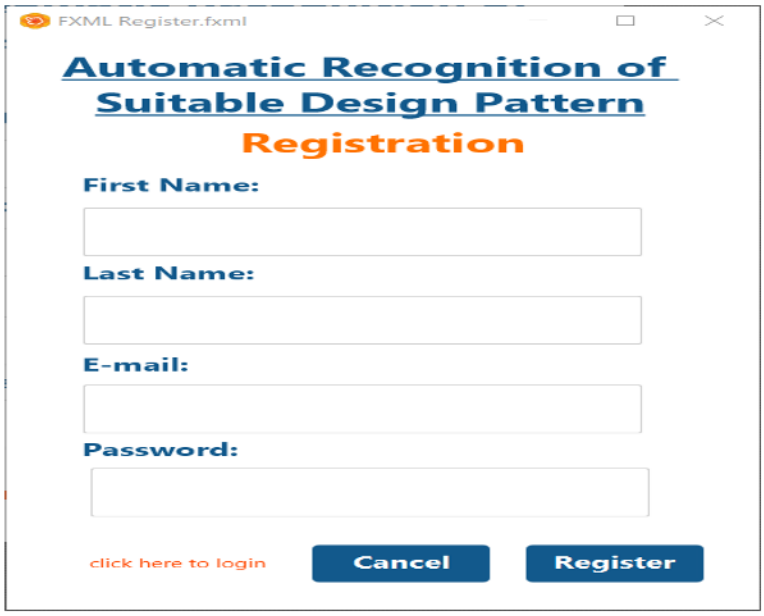


Figure 15: Register Form

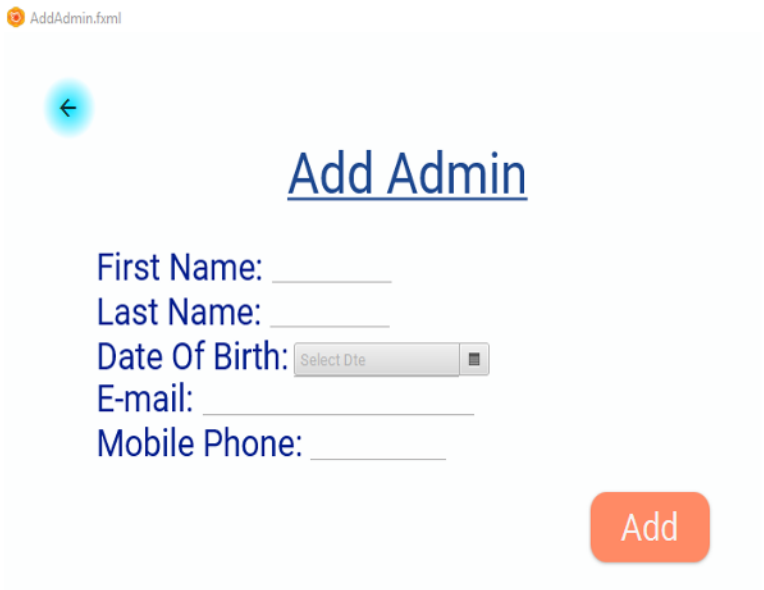


Figure 16: Add Admin

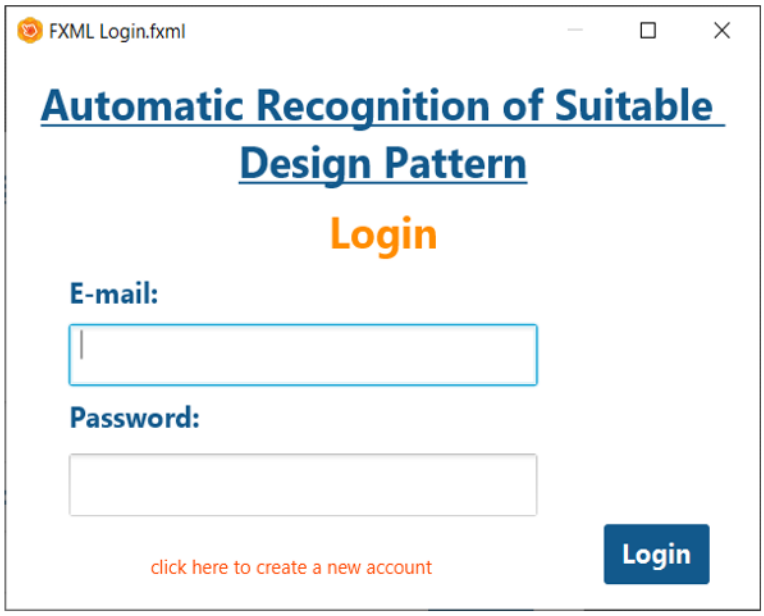


Figure 17: Log In Form

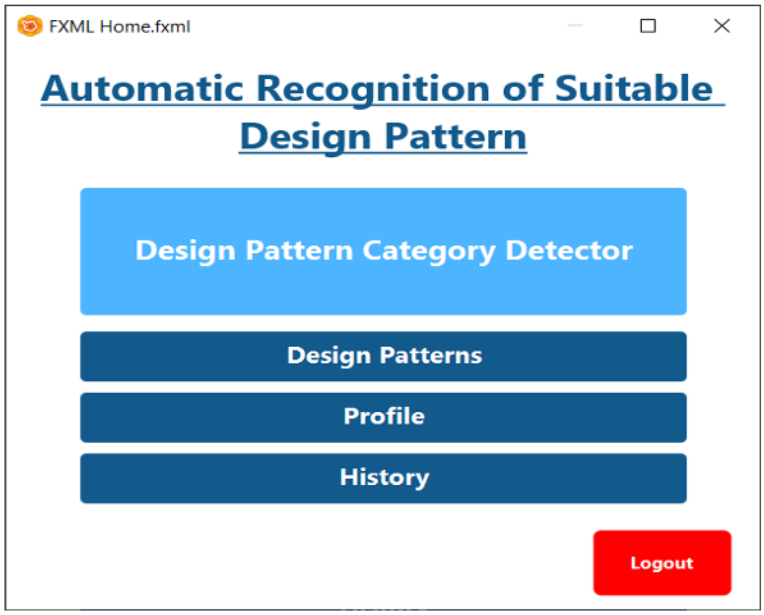


Figure 18: Homepage

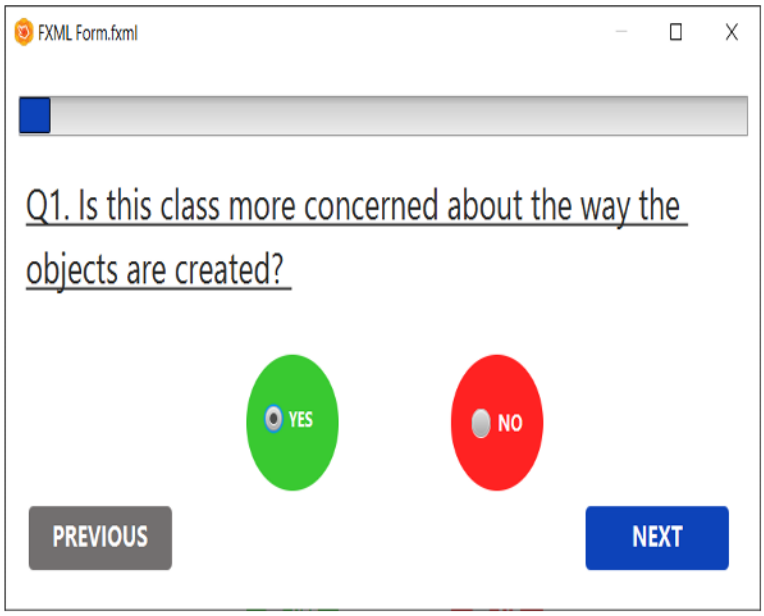


Figure 19: Form(1)

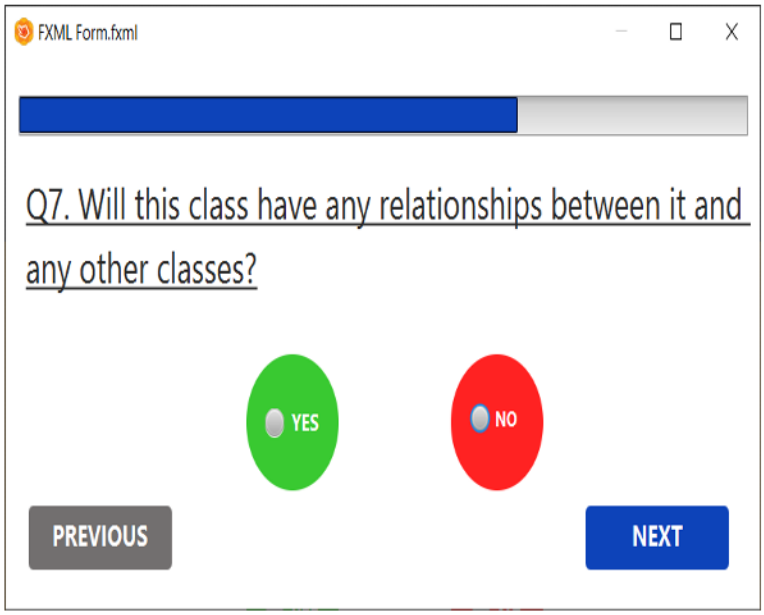


Figure 20: Form(2)

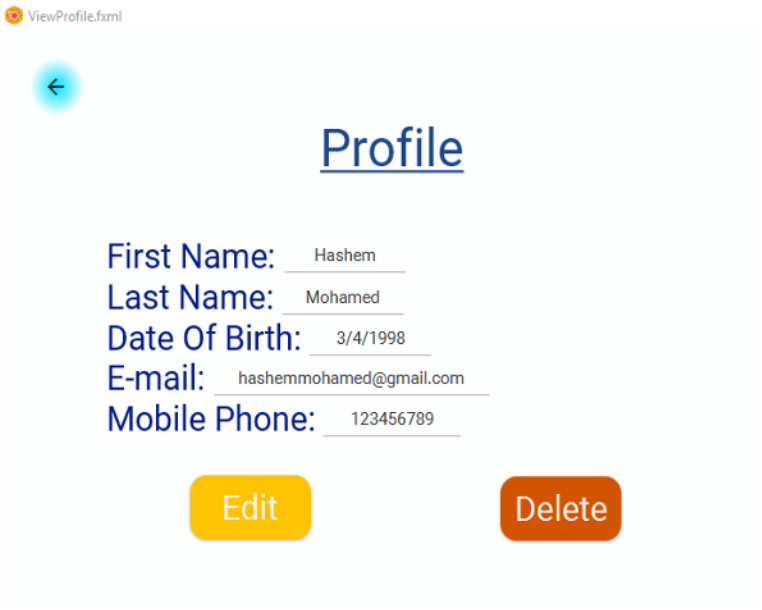


Figure 21: Profile

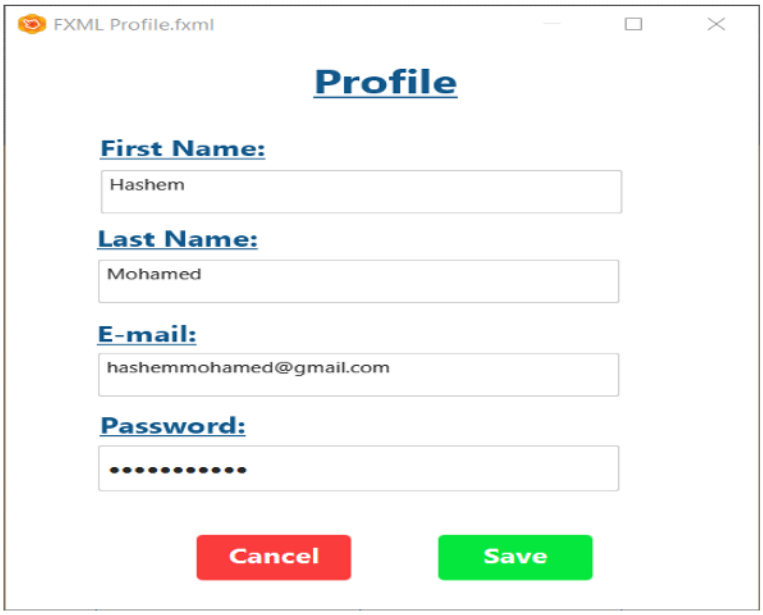


Figure 22: Edit Profile

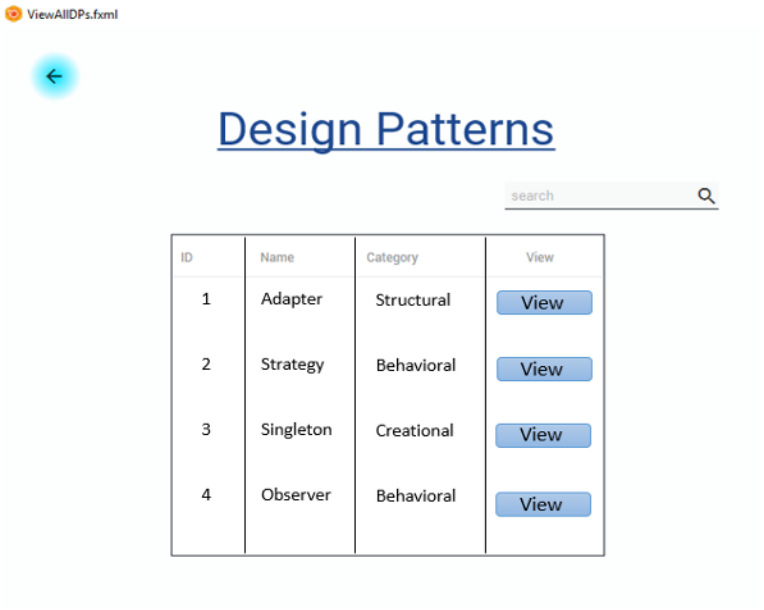


Figure 23: All Design Patterns

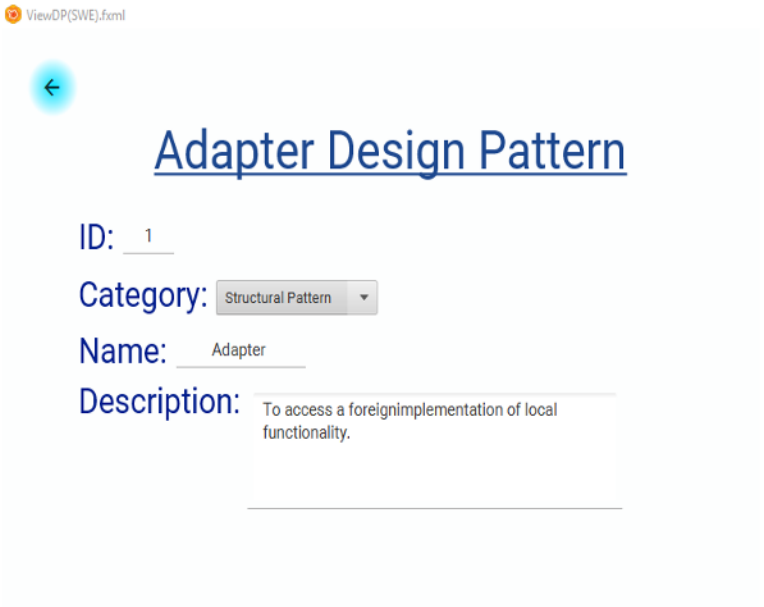


Figure 24: Design Pattern

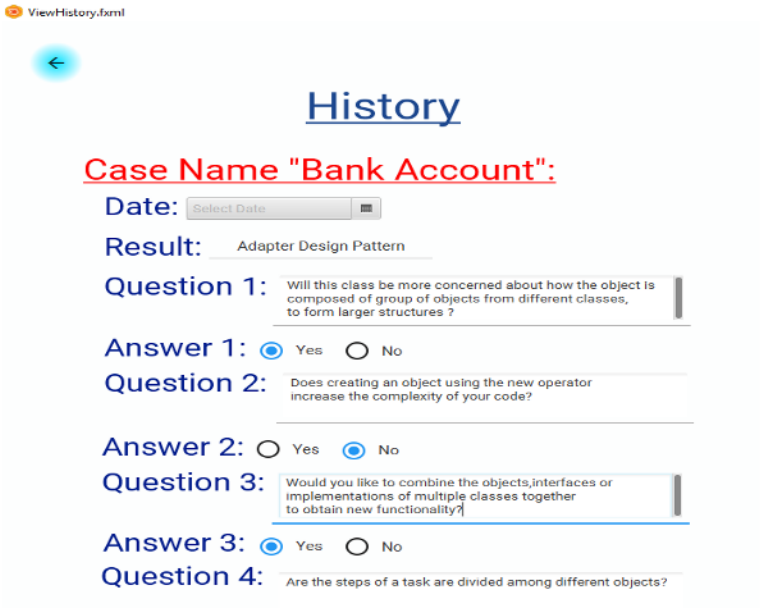


Figure 25: History

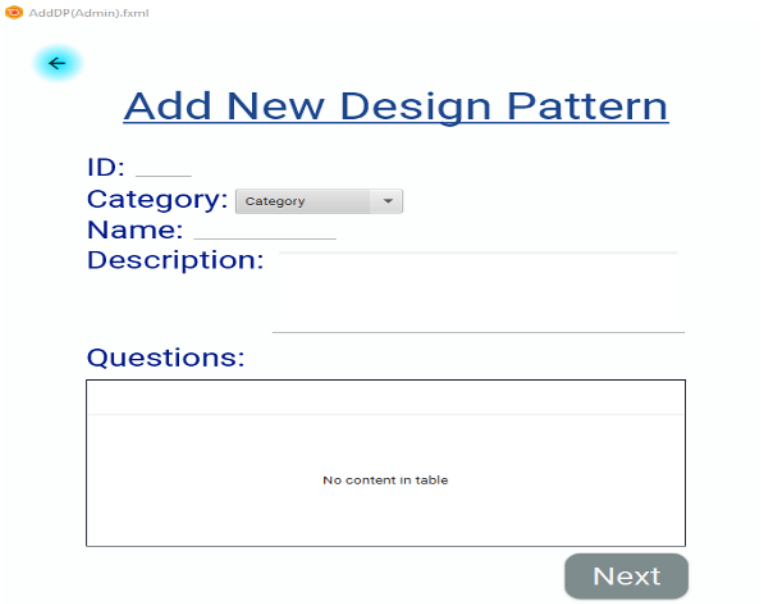


Figure 26: Add New Design Pattern

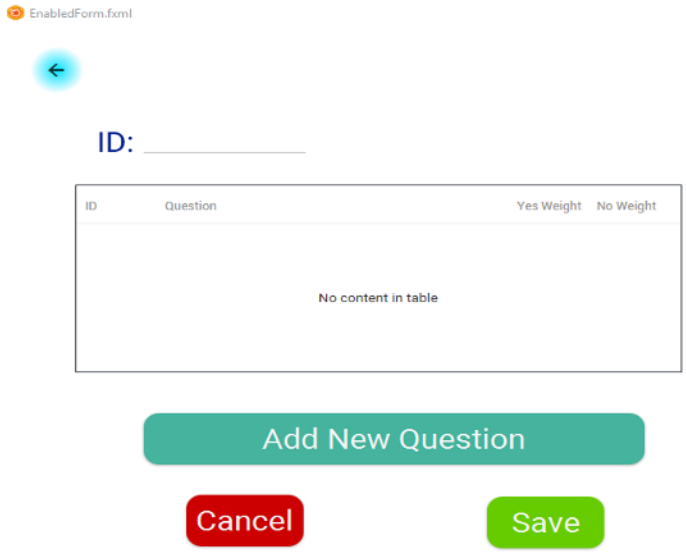


Figure 27: Add Questions to Design Patterns

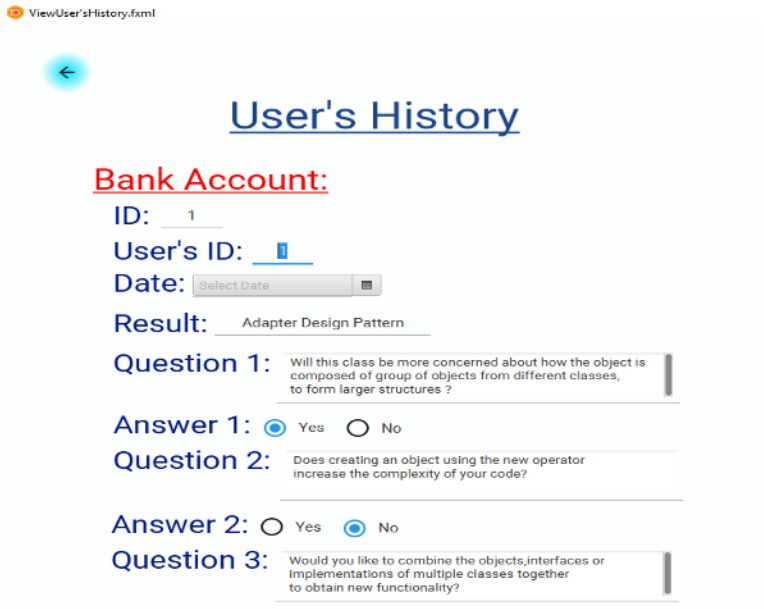


Figure 28: User's History

6.3 Screen Objects and Actions

There are two types of users in the system, software engineers and system administrators. Software engineers can register to the system using the registration form as in Fig.15. However, administrators can not register their accounts in the system but current administrators can add them using the form in Fig.16.

As shown in Fig.17, the users have the ability to log in to his account and access all the main functions of the system. These main functions depends on the user type. Fig.18, represents the homepage of the software engineer.
The first option in this homepage is the design pattern recommendation which generates the questions form for the user as shown in Fig.19 and 20.
The second option is displaying all design patterns available in the system as displayed in Fig 23. By clicking "View" button, each pattern can be viewed in details as in Fig.24.
Fig.21 represents the third option which is viewing the personal information of the SWE and have the ability to edit any of his/her information as in Fig.22.
The last option available to the user is the history display which shows all the previous detection processes that the SWE did. He/She can view a specific history record and it will be displayed as in Fig.25.

If the logged in used is an administrator, he will have the ability to view all design patterns, edit, delete or add new design patterns. The form in which a new design pattern can be added is represented in Fig.26. The "Next" button transfers the user to Fig.27 in order to insert the questions and the answers' weights for each. Moreover, the administrator can view all user's history as in Fig.28.

7 Requirements Matrix

Req. ID	Req. Type	Req. Name	Module	Status	Where In Document
3.1.1	Required	Log In	Software Engineer	Completed	Class Diagram, Activity Diagram
3.1.2	Required	Log Out	Software Engineer	In Progress	Class Diagram, Activity Diagram
3.1.3	Required	Register	Software Engineer	Completed	Class Diagram
3.1.4	Required	Update Profile	Software Engineer	Completed	Class Diagram
3.1.5	Required	View Profile	Software Engineer	Completed	Class Diagram, Activity Diagram
3.1.7	Required	View Homepage	Software Engineer	Completed	Class Diagram, Activity Diagram
3.1.9	Required	View DP Description	Software Engineer	In Progress	Class Diagram
3.1.10	Required	View All DPs	Software Engineer	In Progress	Class Diagram
3.1.11	Optional	Print Result	Software Engineer	Completed	Class Diagram, Activity Diagram
3.1.12	Required	View All History Records	Software Engineer	In Progress	Class Diagram, Activity Diagram
3.1.13	Required	View History Record	Software Engineer	In Progress	Class Diagram
3.1.14	New Requirement	Delete History Record	Software Engineer	In Progress	Class Diagram
3.1.15	New Requirement	Delete All History Records	Software Engineer	In Progress	Class Diagram
3.1.16	Required	View Recommendation Result	Software Engineer	Completed	Class Diagram, Activity Diagram, Sequence Diagram
3.1.17	Required	Answer Questions	Software Engineer	Completed	Class Diagram, Activity Diagram
3.2.1	Required	Update Profile	System Administrator	In Progress	Class Diagram
3.2.2	Required	Add Admin	System Administrator	In Progress	Class Diagram
3.2.3	Required	Update Users' Profile	System Administrator	In Progress	Class Diagram
3.2.4	Required	View Users'	System Administrator	In Progress	Class Diagram
3.2.5	Required	View All Users	System Administrator	In Progress	Class Diagram
3.2.6	Required	Delete User	System Administrator	In Progress	Class Diagram
3.2.7	Required	Insert Category	System Administrator	In Progress	Class Diagram
3.2.8	Required	Update Category	System Administrator	In Progress	Class Diagram
3.2.9	Required	View All Categories	System Administrator	In Progress	Class Diagram
3.2.10	Required	View Category	System Administrator	In Progress	Class Diagram
3.2.11	Required	Delete Category	System Administrator	In Progress	Class Diagram
3.2.12	Required	Insert Design Pattern	System Administrator	In Progress	Class Diagram
3.2.13	Required	Update Design Pattern	System Administrator	In Progress	Class Diagram
3.2.14	Required	View Design Pattern	System Administrator	In Progress	Class Diagram
3.2.15	Required	View all Design Patterns	System Administrator	In Progress	Class Diagram
3.2.16	Required	Delete Desgin Pattern	System Administrator	In Progress	Class Diagram
3.2.17	Required	View All Users' History Records	System Administrator	In Progress	Class Diagram
3.2.18	Required	View User's History Record	System Administrator	In Progress	Class Diagram
3.2.19	New Requirement	Delete Users' History Record	System Administrator	In Progress	Class Diagram
3.2.20	New Requirement	Delete All History Records	System Administrator	In Progress	Class Diagram
3.3.1	Required	Retrieve Metrics Weights	Weight Calculation	In Progress	Class Diagram
3.3.2	Required	Compute Final Metric Weight	Weight Calculation	In Progress	Class Diagram, Process Diagram
3.4.1	Required	Encryption/Decryption	Security	In Progress	Class Diagram
3.4.2	Required	Emails Matching	Security	In Progress	Class Diagram

Figure 29: Requirements Matrix

8 APPENDICES

Table 5: Singleton Design Pattern

Function	To enclose a global resource.
Intent	Make sure the only one instance is created for a class with a global access point to it.
Objective	Prevent creating multiple instances for a specific class.
Disadvantage	* Singleton client code is difficult to be unit tested, Needs a specialized treatment in the case of multi-threading to prevent the creation of many singleton objects. * May cover bad code designs, Breaks the rule of "Single Responsibility" principle.
When to use	Creating a single instance that can be accessed by all clients, Providing a harsh control on global variables.

Table 6: Prototype Design Pattern

Function	To establish objects by sample.
Intent	* Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. * Co-opt one class instance for all future instances to be use d as a breeder. * The new operator considered harmful.
Objective	To create a clone (Copy) of an object without getting your code coupled with the class of that object.
Disadvantage	It could be very tricky to clone complex objects with circular references.
When to use	* When a system should be independent of how its products are formed, composed, and performed. * When defining the classes to be built at run-time, such as: by dynamic loading. * To avoid building a class hierarchy of factories that equivalents the class hierarchy of products. * When a class instances can have one of only some various case combinations. Installing and cloning a corresponding number of prototypes may be more suitable than manually installing the class, with the proper state.

Table 7: Abstract Factory "Kit" Design Pattern

Function	To build objects by standard and preconfigured type.
Intent	* Provide an interface for building families of linked or dependent objects without assigning their specifying classes. * A hierarchy encapsulating: many possible "platforms" and a suite of "products" being built. * The new operator considered harmful.
Objective	To prevent using lots of if case statements in your portable application so it can support many platforms including: database, operating system, windowing system, etc.
Disadvantage	The code may become more complex than it should be, as many new interfaces and classes are added together with the pattern.
When to use	* A system should be independent of how its products are established, consisted of, and appeared. * One of several families of products must be configured with a system. * A family of related product objects is planned to be used together, and this constraint wants to forced. * A class library of products needs to provide, and wants to detect only their interfaces, not their implementations.

Table 8: Builder Design Pattern

Function	To split the construction of a complex object from its representation, in order to create different representation in the same construction process.
Intent	Analyze a complex representation, establish one of several targets.
Objective	* To provide a different way to construct complicated objects. *To construct an different fixed objects using the exact object building process.
Disadvantage	The code's overall complexity increases as the pattern needs multiple new classes to be developed.
When to use	* The algorithm for establishing a complex object should be independent of the parts that make up the object and how they're assembled. *The structuration process must allow different representations for the object that's built.

Table 9: Composite Design Pattern

Function	Pass over a recursive tree-structure uniformly.
Intent	Construct tree structures of objects to allow clients deal with the structures like individual objects.
Objective	Prevent differentiating between the primitive (simple) objects and composite (complex) objects, which are collections of primitive objects, even when they act the same most of the time, to reduce application complexity.
Disadvantage	When the functionality of classes has many differences, it becomes difficult to create a common interface.
When to use	*When the application has hierarchical tree-structure. *When the application wants to set a fixed functionality through the whole structure, deal with composites (complex) and individual (simple) objects uniformly.

Table 10: Bridge "Handle/Body" Design Pattern

Function	Split the hierarchies of abstraction (user interface) and implementation.
Intent	Split the abstraction from its implementation so they can change independently.
Objective	Prevent the hierarchy from growing exponentially that happens by initiating a concrete class after each action made by the client that leads to difficulty in editing, extending, and reusing abstractions and implementations independently This can be done by switching from inheritance to composition.
Disadvantage	Code Complexity.
When to use	*Cover the implementation of an abstraction from clients. *Share an implementation among multiple objects and hide this from the client. *Abstractions and their implementations should be extensible by sub classing. *Implementations switching during runtime without affecting the client.

Table 11: Proxy "Surrogate" Design Pattern

Function	Build a wrapper to hide the object's complication from the client.
Intent	Create a new object to act as a substitute for the original object to control its access, by sending the request from the new object to the original object.
Objective	Prevent the creation of high weight objects until a request is received.
Disadvantage	*Complicated codes. *Delayed service response.
When to use	*When there is a need for a practical reference to an object. *Common cases: -Remote proxy (Ambassador): a local model for an object is initiated in another address space. -Virtual proxy (lazy loading): construct objects with heavy weight by request. -Protection proxy (Access control): add a security layer by managing different access rights of objects. -Smart reference: replacing empty pointers that performs additional actions when an object is accessed, discard heavy weight objects when they are not in use.

Table 12: Flyweight Design Pattern

Function	Avoid excessive creation of global resources.
Intent	*Substitute heavy-weight widgets with light-weight gadgets. *Fit several objects into the available memory finegrained by sharing common states efficiently.
Objective	Reduce the design cost in terms of memory consumption and run-time overhead (performance).
Disadvantage	*Code Complexity. *More CPU cycles due to the recalculation of data whenever it's requested.
When to use	*When the objects have common state (intrinsic state) that can be extracted and shared among different objects. *When large number of objects is used. *When the cost of storage is high. *When the application does not rely on object identity.

Table 13: Decorator Design Pattern

Function	Adding extra behaviors to an object.
Intent	Dynamically assign extra responsibilities to an object. Substitute subclassing to expand functionalities.
Objective	Make an alternative for static inheritance in order to add behaviors to specific objects through runtime.
Disadvantage	Difficult to ignore stack order while implementing, Difficult to remove a specific wrapper from stack.
When to use	Adding to objects extra behaviors through runtime, When it's hard to expand object's behaviors through inheritance.

Table 14: Facade Design Pattern

Function	To hide the system's complexities and provide the client with an interface from where the client can access the system.
Intent	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Objective	To implement a simplified interface to the universal functionality of a complicated subsystem.
Disadvantage	Can be a god object coupled to all classes of an app.
When to use	* This pattern is suitable if we have a complex system that we want to expose to clients in a simplified way. The aim is to mask internal complexity from the outside behind a single interface. * It also decouples the code that uses the system from the details of the subsystems, making it easier for the system to be modified later.

Table 15: Adapter Design Pattern

Function	To access a foreign implementation of local functionality.
Intent	*Convert a class interface to another expected client interface. *Adapter allows classes to work together because of incompatible interfaces that could no otherwise.
Objective	To reuse any component without being worried about compatibility between it and the system you are currently developing.
Disadvantage	The code's overall complexity increases because a set of new interfaces and classes needs to be introduced. Sometimes just changing the service class to match the rest of your code is simpler.
When to use	When a class you want to use does not fulfill the interface requirements.

Table 16: Iterator "Cursor" Design Pattern

Function	Traverse the container (collection) and access its elements consecutively.
Intent	Abstract the traversal of hugely different data structures in order to define algorithms that can communicate transparently with each other.
Objective	* Support a collection traversal to "full object status." * Standard library abstraction of C++ and Java allowing the decoupling of collection classes and algorithms. * Access the aggregate object's elements consecutively without revealing its underlying representation.
Disadvantage	* If your application only operates with simple collections, implementing this pattern can be an overkill. * It may be less effective than the direct use of elements of certain specialized collections.
When to use	* When accessing the contents of an aggregate object without displaying its internal representation. * Supports multiple aggregate object traversals. * Providing a standard framework for traversing various aggregate structures (supporting polymorphic iteration).

Table 17: Chain of Responsibility Design Pattern

Function	Enable potential handlers to transfer requests through the chain until one of them handles requests. Allow multiple entities to handle requests in the concrete classes of the receivers without coupling sender class.
Intent	* Do not couple the recipient of the request to the receiver by allowing more than one object the opportunity to handle the request. *Chain the receiving objects and move the request through the chain until it is handled by the object. *Launch-and-leave requests with a single storage pipeline containing a large number of handlers. *It is an object-oriented linked list which traverse recursively.
Objective	Prevent hard wiring handler/processing element/node objects relationships and precedence, or mappings of request-to-handler as there's always a stream of requests that needs to be handled.
Disadvantage	Not handling all requests.
When to use	* A request can be handled by more than one object. * Issuing a request to one of many objects, without identifying the receiver clearly. * The objects which handle a request must be identified dynamically.

Table 18: Command "Action/Transaction" Design Pattern

Function	To isolate the request from the execution.
Intent	* The requests are encapsulated as objects and helps parameterize the client with different queue/requests/log requests. * An object-oriented callback. * Support "invocation of an object method" to the full status of the object.
Objective	The request is turned itself into an object while knowing nothing about the requested operation or the request receiver.
Disadvantage	Since there is a new layer between senders and receivers is being introduced, the code might become more complicated.
When to use	* When issuing requests into objects while knowing nothing about the requested operation. * When undo is supported. * Implementation of menus which is easy with Command objects because it parameterize objects. * Structure a system based on basic operations through high-level operations. * Help recording changes so that in the event of a system crash they can be reapplied. * Specify, queue, and run requests at various times.

Table 19: State "Objects for States" Design Pattern

Function	Shift among different implementations of the whole functionality.
Intent	* Authorize an instance to change its behavior when a change happen in its internal state. The class of the instance will appear to be changed. * A state machine that is concerned about object-oriented. * Collaboration,polymorphic wrapper, wrapper.
Objective	It aids the monolithic instance's behavior to change based on its state by creating a class to each state, instead of implementing all behaviors on its own.
Disadvantage	* Implementing the pattern can be excessive if machine's state consist of few states or barely changes.
When to use	When an instance behavior rely on it state, and it needs to change its behavior at run time relying on that state.

Table 20: Memento "Token" Design Pattern

Function	Is to capture and object's internal state so that the object can be restored to this state later (Storing a snapshot).
Intent	Lets you capture and restore the latest state of an instance without revealing its implementation details.
Objective	It prevents the loss of an instance state by creating a snapshot of the instance and restores it back to its previous state when it faces any failures.
Disadvantage	* The majority of the dynamic programming languages can't assure that the state within the memento design pattern remains unaffected. * It may cause RAM consumption if user creates snapshot regularly.
When to use	* It aids the user to recover and restore from any failures because it can revert to its previous state. * When a direct interface gathering the state of an instance would crack the instance encapsulation and expose implementation details.

Table 21: Interpreter Design Pattern

Function	Defining a grammatical description for a certain language and implements an interpreter to handle this grammar.
Intent	* Provided a language, determine a n illustration for its grammar among the interpreter that applies the illustration to interpret sentences in a certain language. * Mapping a certain domain to a language, then the language into certain grammar and then the grammar into a object oriented hierarchical design.
Objective	In a well-defined and understood domain, a class of problems happens repetitively. If the domain was defined by a "language," an "engine" interpretation would easily fix problems.
Disadvantage	Complicated grammar are hard to preserve. The Interpreter design pattern illustrates at least one class for each and every rule in the grammar. Because of that the grammar that contains many rules are hard to be managed and preserved.
When to use	* The optimum usage is when the grammar is not complicated. For complicated grammars, the hierarchy of each class becomes big and hard to manage. Using tools would be better in cases like that such as parser generators. * The perfect usage is when efficiency is not the major matter. All the efficient interpreters are not directly implemented by interpreting parse trees, first they get translated into different form. as an example, regular expressions are converted to state machines. After that with the Interpreter design pattern the translator gets implemented.

Table 22: Template Method Design Pattern

Function	* Illustrates the skeleton of a certain method in an activity. setting aside some procedures to subclasses. * It lets the subclasses reconsider certain procedures of a method without any change in the method's structure.
Intent	Highlight and assure the execution of universal process.
Objective	Prevents duplication by excluding the duplicate effort that happens when two various elements have the similarities but determine no reuse of typical implementation or interface.
Disadvantage	* Implementing the pattern can be excessive if machine's state consist of few states or barely changes. * Liskov Substitution Principle may get violated by forcibly putting an end to a regular step implementation by a subclass. * Some of the clients may face some limitations by implemented skeleton of a method or an algorithm.
When to use	* Implementing the similar sections of a method once and authorize it to subclasses so it can implement the behaviors that may alter. * When there is a similar behavior among different subclasses that should be determined and localized in one class to avoid duplication of code. * Authorize at which exact points sub classing is permitted.

Table 23: Visitor Design Pattern

Function	To select discrete functionality through the type of object.
Intent	* it illustrates a process to be implemented on the fundamental of the object's structure. * Creates a new process without any change in the fundamentals of the classes. *The usual and typical approach to recover lost type information on which it operates. *forming the most accurate decisions depending on the two objects type. *Double dispatch.
Objective	*Prevents the pollution of node classes caused by many unrelated activities that require to be executed on node instances in a different structure. * Avoids the need to query the type of every node and to cast the pointer to the appropriate type ahead of executing the needed.
Disadvantage	* When updating all visitors each time a new class is added or removed from the factors hierarchy. * Visitors might find it hard necessary access to the private attributes and methods of the factors that they should interact with.
When to use	* When instances structure is shared by many applications that depends on them. * It lets you keep linked activities together through defining each one of them in one class.

Table 24: Mediator Design Pattern

Function	* It encapsulates all interconnections and serves as a coordination hub. * Extract all classes relationships into a separate class, separating from the rest of the components any changes to a specific component. * Responsible for controlling and organizing client communications.
Intent	* Identifies an object encapsulating the interaction of a group of objects. * Design an intermediate step for many peers to be decoupled. * Promoting many-to-many relationships to "full object status" between communicating peers. * Stopping objects from directly referring to each other, it supports loose coupling and allows to vary their communication independently.
Objective	* Reduce direct reference between objects as it helps creating loosely-coupled interaction between them. * Facilitate the communication between objects in a way where the objects are unaware of other objects' presence. Instead of being coupled to hundreds of other objects, they rely on a single mediator class. * Prevent creating a complex system caused by direct communication when multiple objects must interact with each other to handle the request.
Disadvantage	A mediator may transform into a "God Object" over time.
When to use	* In well-defined yet complex ways, a group of objects interact and it is impossible to reuse an object as it refers to many objects and interacts with them. * Behavior among several classes should become flexible without much sub-classing and the inter-dependencies generated are unstructured and hard to understand.

Table 25: Observer "Dependents/Publish Subscribe" Design Pattern

Function	To synchronize state change.
Intent	* Defines a one-to many dependency between objects so that all its dependents are automatically notified and updated when on e object changes state. * Encapsulate the core components in a subject concept and the components of the UI (User Interface) in an Observer hierarchy. * The "View" part of "Model-View- Controller".
Objective	To prevent insufficient scalability in a big self-contained design as new monitoring or graphing specifications are levied.
Disadvantage	Subscribers are informed in random order.
When to use	When having a system design in which several entities are involved in any probable update to some special second entity object.

9 References

References

[1] M. E. Abeer Hamdy, "Topic modelling for automatic selection of software design patterns," *JSW*, vol. 13, pp. 260–268, 2018.

[2] A. K. Dwivedi, A. Tirkey, R. B. Ray, and S. K. Rath, "Software design pattern recognition using machine learning techniques," pp. 222–227, 2016.

[3] M. Elallaoui, K. Nafil, and R. Touahni, "Automatic transformation of user stories into uml use case diagrams using nlp techniques," *Procedia computer science*, vol. 130, pp. 42–49, 2018.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[5] A. Hamdy and M. Elsayed, "Automatic recommendation of software design patterns: Text retrieval approach." *JSW*, vol. 13, no. 4, pp. 260–268, 2018.

[6] H. Herchi and W. B. Abdessalem, "From user requirements to uml class diagram," *arXiv preprint arXiv:1211.0713*, 2012.

[7] F. Palma, H. Farzin, Y. Guéhéneuc, and N. Moha, "Recommendation system for design patterns in software development: An dpr overview," in *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, June 2012, pp. 1–5.

[8] A. Shevts, *Dive Into Design Patterns*. Refactoring.Guru, 2018.