Design Document

NoSQL Project

**Class:** IT2A

**Team:** 01  
**GitHub:** <https://github.com/Athlon007/CompetentHelpDesk>

**Written by:**

* Fatmanur, Vardar, 652369
* Konrad Figura, 680886
* Maria Diaconescu, 683265
* Rodrigo Bange, 684006
* Sara Eftekhar Azam, 660310

Table of Contents

[Introduction 1](#_Toc119075328)

[1.Methods used during the development process 2](#_Toc119075329)

[1.1. Entity relationship diagram 2](#_Toc119075330)

[1.2. UML Diagram 3](#_Toc119075331)

[1.2.1. The Model Layer 3](#_Toc119075332)

[1.2.2. The Data Access Layer 4](#_Toc119075333)

[1.2.3. The Logic Layer 4](#_Toc119075334)

[1.2.4. The User Interface layer 5](#_Toc119075335)

[1.3. UML diagrams based on division of tasks 6](#_Toc119075336)

[1.3.1. Adding employees and ticket transfer 6](#_Toc119075337)

[1.3.2. Dashboard and tickets and incident filtering 7](#_Toc119075338)

[1.3.3. Create, read, update, delete tickets and escalate tickets 8](#_Toc119075339)

[1.3.3. Display the tickets to the employee, close tickets and searching through tickets 9](#_Toc119075340)

[1.3.4. Login, creating incidents and ticket archiving 10](#_Toc119075341)

[Konrad Figura - Design Choices 11](#_Toc119075342)

[Use of StatusStruct 11](#_Toc119075343)

[TicketsDAO use of BsonDocument, instead of Ticket as a generic 11](#_Toc119075344)

[Ticket Escalation as a field of the Ticket document 11](#_Toc119075345)

[Use of Newtonsoft.Json 11](#_Toc119075346)

[Inability to edit dates in submitted ticket 11](#_Toc119075347)

[Data transfer objects for InsertTicket and UpdateTicket 11](#_Toc119075348)

[Rodrigo Bange – Design Choices (Dashboard & Incident Filtering) 12](#_Toc119075349)

[Dashboard 12](#_Toc119075350)

[Long value to int conversion 12](#_Toc119075351)

[Incident Ticket Filtering (Individual part) 12](#_Toc119075352)

[Service Ticket Filtering 12](#_Toc119075353)

[Logic behind Ticket Count 12](#_Toc119075354)

[Logic behind Ticket Filtering 13](#_Toc119075355)

[Circular Progression Bar 13](#_Toc119075356)

[Faster Panel & Table Layout Panel 13](#_Toc119075357)

[Rounded Panels, buttons and custom icons and logos. 13](#_Toc119075358)

[**Sara Eftekhar Azam – Design Choices** 14](#_Toc119075359)

[Searching 14](#_Toc119075360)

[Close Ticket 14](#_Toc119075361)

[Close Ticket Search 14](#_Toc119075362)

[**Fatmanur Vardar – Design Choices** 15](#_Toc119075363)

[Adding Employees 15](#_Toc119075364)

[Ticket Transfer 15](#_Toc119075365)

# Introduction

In the present design document, we want to explain the implementation of the requirements from the rubrics provided for the service management application. It includes the diagrams that were used in the development process and the description of the design choices made during development by each team member.

The first step in the progress of the application was the creation of the Entity Relationship Diagram. In this step the members of the team included the entities and relationships that they were using according to the requirements that they were assigned for implementation.

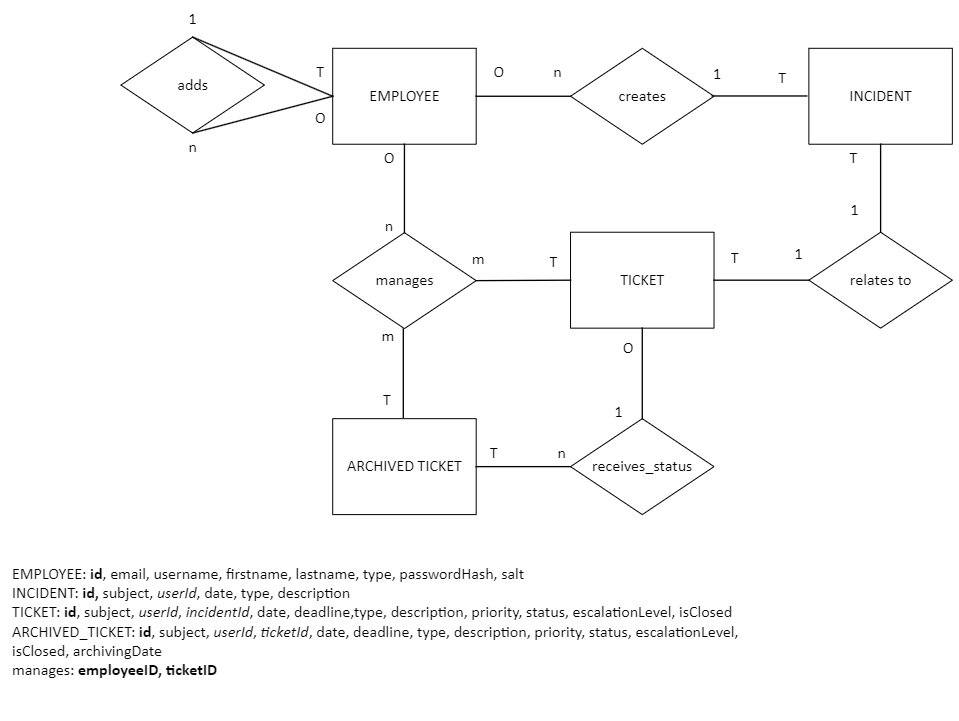
In the next step the team created the UML in order to describe the models used in the application, which refers to the description of the multiple layers that communicate within the application and the relationships between them.

In order to be included in the present document, the UML diagram was split in different sections that contain the layers of the application that contain the parts from the various layers on which each team member worked.

In the chapter from this document where the design choices are explained is detailed how the functionalities of the application were implemented. To the implementation it contributed with great importance the design decisions made during the development process by each team member because the division of tasks was clearly set from the beginning of the project.

# 1.Methods used during the development process

## 1.1. Entity relationship diagram



The entities chosen for modelling the data from the NoSQL database are the following: EMPLOYEE, INCIDENT, TICKET, ARCHIVED TICKET.

The first relationship was named *creates* and links the entity Employee to the entity INCIDENT. It refers to the fact that one employee can register a new service call or incident if they are of the type regular employee.

The second relationship named *relates to* exists between the entities INCIDENT and TICKET. It exists in the diagram because from one incident a ticket can be created by a service desk employee.

The next relationship is called *manages* and it links the entity Employee with the entity TICKET. This relationship exists when one employee is of the type service desk employee and they create, update, or delete a ticket.

Another relationship was determined between the entities TICKET and ARCHIVED TICKET. It means that one ticket, after a certain interval, is archived.

# 1.2. UML Diagram

The application created contains the following layers: the Model layer, which contains the classes that represent the data, the data access layer which is used for defining methods that perform CRUD operations on the database, the logic layer which represents the communication layer between the Data Access Layer and the User Interface which is communicating with the user.

## 1.2.1. The Model Layer

The Model layer include the classes, enumerations and structs: Employee, EmployeeType, RNG, HashedPasswordWithSalt,PasswordWithSaltHasher, Incident, IncidentTypes, Ticket, TicketPriority, TicketStatus, TicketdateTransfer, TicketEmployeeTransfer, TicketEnumsTransfer, TicketTextTransfer, ArchivedTicket which is the model of the data related to archived tickets.

Employee is the model for the data about employees.

EmployeeType enumeration contains the types of employees: Regular, Service Desk, Specialist and UpperManagement.

RNG is a class used for login with the purpose to create cryptographic keys,

HashedPassword WithSalt is used to store the encrypted passwords.

PasswordWithSaltHasher is a class utilized to create hashed passwords using a specific hashing algorithm.

Incident is used to represent the data related to incidents.

IncidentTypes enumeration stores the types of incidents.

Ticket is used to represent the data about tickets.

TicketPriority enumeration contains the priorities that can be assigned to a ticket:ToBeDetermined, Low, Medium, High

TicketStatus enumeration includes the statuses that a ticket can have: Open, PastDeadline, Unresolved, Solved.

TicketDateTransfer is a struct used for setting and updating the ticket registration date and the interval in which a solution must be provided to the ticket.

TicketEmployeeTransfer is a struct for setting and updating the data about the reporting user and the assigned employee for providing a solution to the ticket.

TicketEnumsTransfer is used for establishing and changing the priority, status and escalation level of the ticket.

TicketTextTransfer is used for establishing and updating the data from the subject and description of the ticket.

ArchivedTicket is the model of the data related to archived tickets.

## 1.2.2. The Data Access Layer

The Data Access Layer contains the methods that are used to create, read, update, and delete records from the database that is used by the application. It also contains the class ConfigFile. These are the following: BaseDAO, EmployeesDAO, IncidentDAO, TicketsDAO, ArchivedTicketsDAO.

All the classes from this layer are inherited from BaseDAO, which retrieves the connection string from a configuration file and then provides the application the access to the database that stores the data that must be managed by the application.

EmployeesDAO includes methods for retrieving the data from the Employees collection based on various criteria, for deserialization of documents and for creating records in the Employees collection.

IncidentDAO contains methods used for retrieving data from the Incidents database, deserialization of documents, creating and deleting data from the Incidents database.

TicketsDAO contains methods for retrieving data from the Tickets collection based on various criteria, counting the documents, creating, updating, and deleting records from the Tickets database.

ArchivedTicket is a class with methods utilized for retrieving data about the tickets that must be archived from the Tickets collection, retrieving data from ArchivedTicketsCollection, serialization and deserialization of documents, creating and deleting records from the ArchivedTickets collection and updating the Tickets collection.

ConfigFile is a class used for reading the data from the configuration file, for providing access to the database.

## 1.2.3. The Logic Layer

The Logic layer is the binding layer between the Data Access Layer and the user Interface Layer. In includes the following classes and structs: Databases, LoginService, EmployeeService, IncidentService, IncidentFilteringService, TicketService, StatusStruct, TicketEscalationService, TicketTransferService, ArchivedTicketService and Error Handler.

Databases is a class that can provide information about the database model.

The EmployeesService contains methods for retrieving data from the Employees collection based on various criteria, deserialization of documents, generating random passwords, and creating records in the Employees database.

LoginService contains methods for creating hashed passwords and checking if the usernames and passwords are correct when the user logs in.

IncidentService is a class with methods for retrieving the data from the Incidents collection, deserialization of documents, creating records in the Incidents database and creating records in the Ticket database from incidents.

IncidentFilteringService contains methods for retrieving the keywords from the filter chosen by the user and filtering the incidents based on the keywords.

TicketService includes methods for querying the Tickets database to retrieve the tickets managed by a certain employee, sorting tickets according to their priority, deserialization of documents, retrieving tickets data using various filters such as status and deadline, counting the documents from the collection, retrieving information about the escalated tickets, creating, updating and deleting records from the Tickets collection, validating the registration and updating of tickets, closing tickets and retrieving information about the closed tickets.

StatusStruct is a struct that was created for storing the validation messages resulted from the operations performed on the Tickets collection.

TicketEscalationService includes methods for checking if a ticket can be escalated and escalating tickets.

TicketEscalationService is a class that was created for transfering the ticket to another employee. It is used for updating the Tickets collection.

ArchivedTicketService is a class with methods for retrieving tickets that must be archived from the Tickets collection, retrieving data from the ArchivedTickets collection, serialization, and deserialization of documents, create records in the ArchivedTickets database and updating the Tickets database.

ErrorHandler is a class that writes the errors that occurred withing the application to a log file.

## 1.2.4. The User Interface layer

The User Interface layer deals with the communication between the user and the application. It is formed of two parts: the Login Screen and Main.

The main contains the elements of the UI used by the user for CRUD operations on the database. It was divided into parts that display the following information to the user:

a dashboard with the current status of all tickets and sections for creating users, registering incidents and tickets, incident management, ticket management and ticket archive.

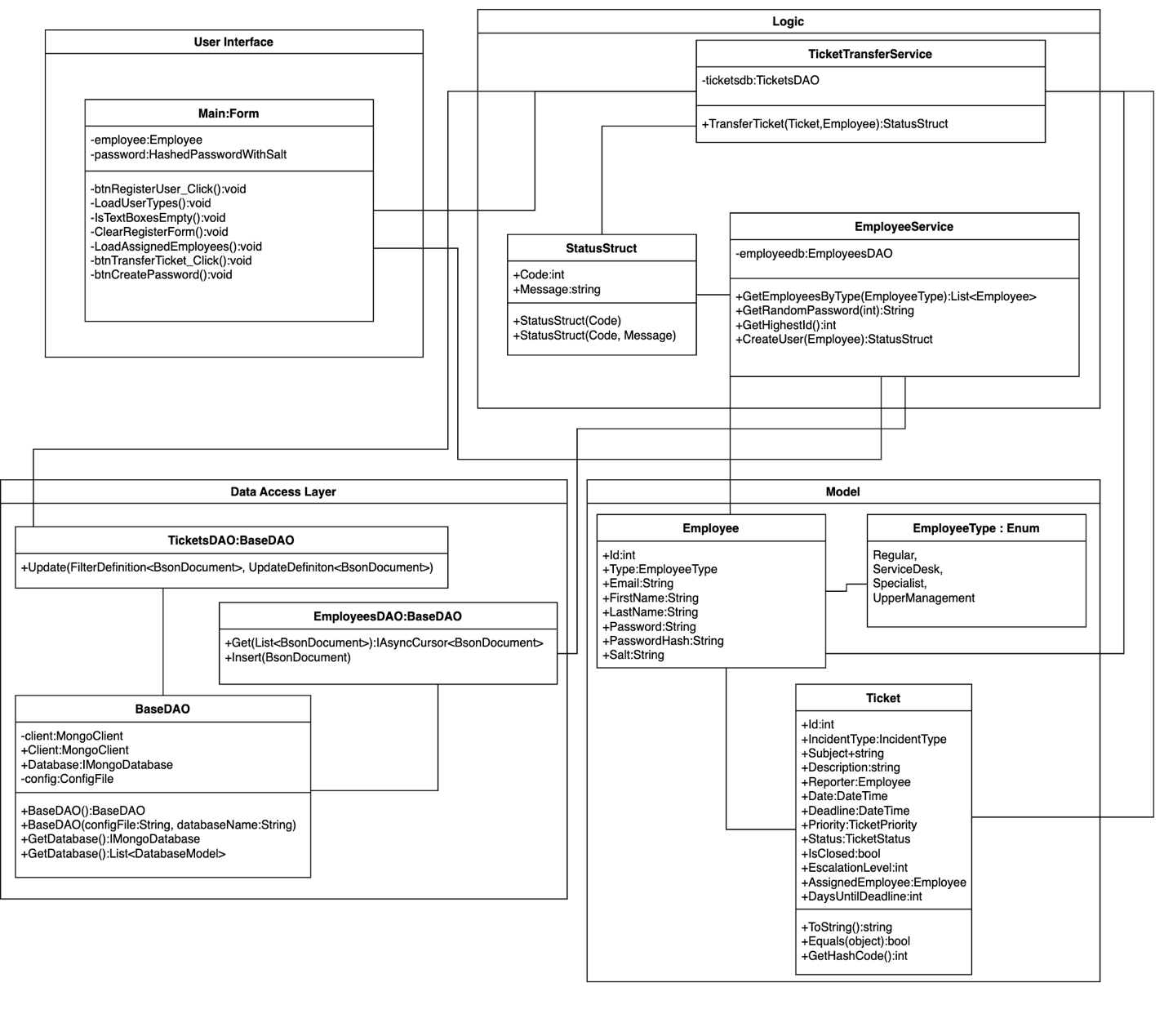
The dashboard displays the amount of tickets which are open, past deadline, closed without resolve and closed in circular progress bars.

In conformity with the UML diagram, the parts of the UI mentioned use the various

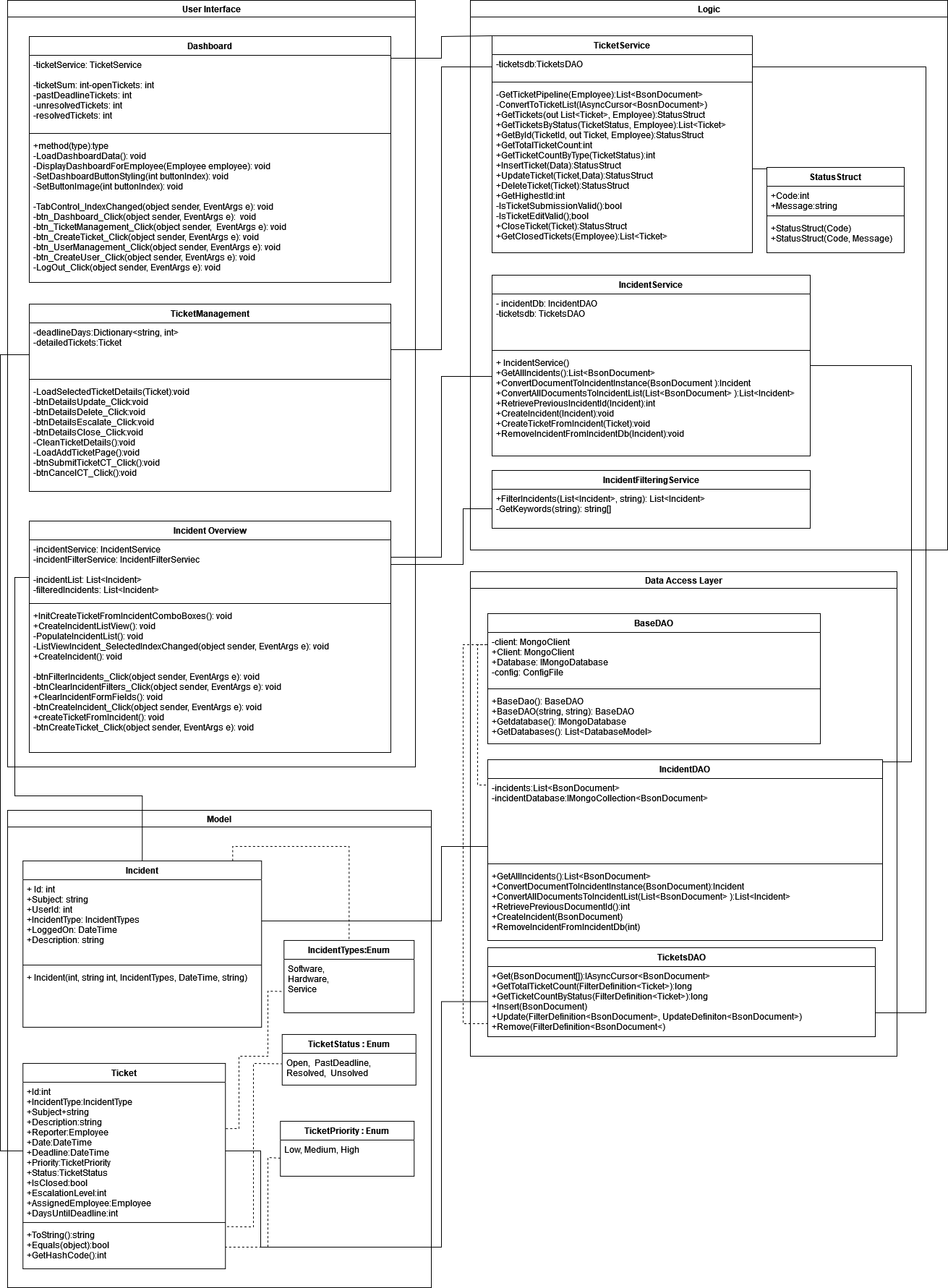
Logic layers for displaying data from the database and sending the required information by user to the database.

# 1.3. UML diagrams based on division of tasks

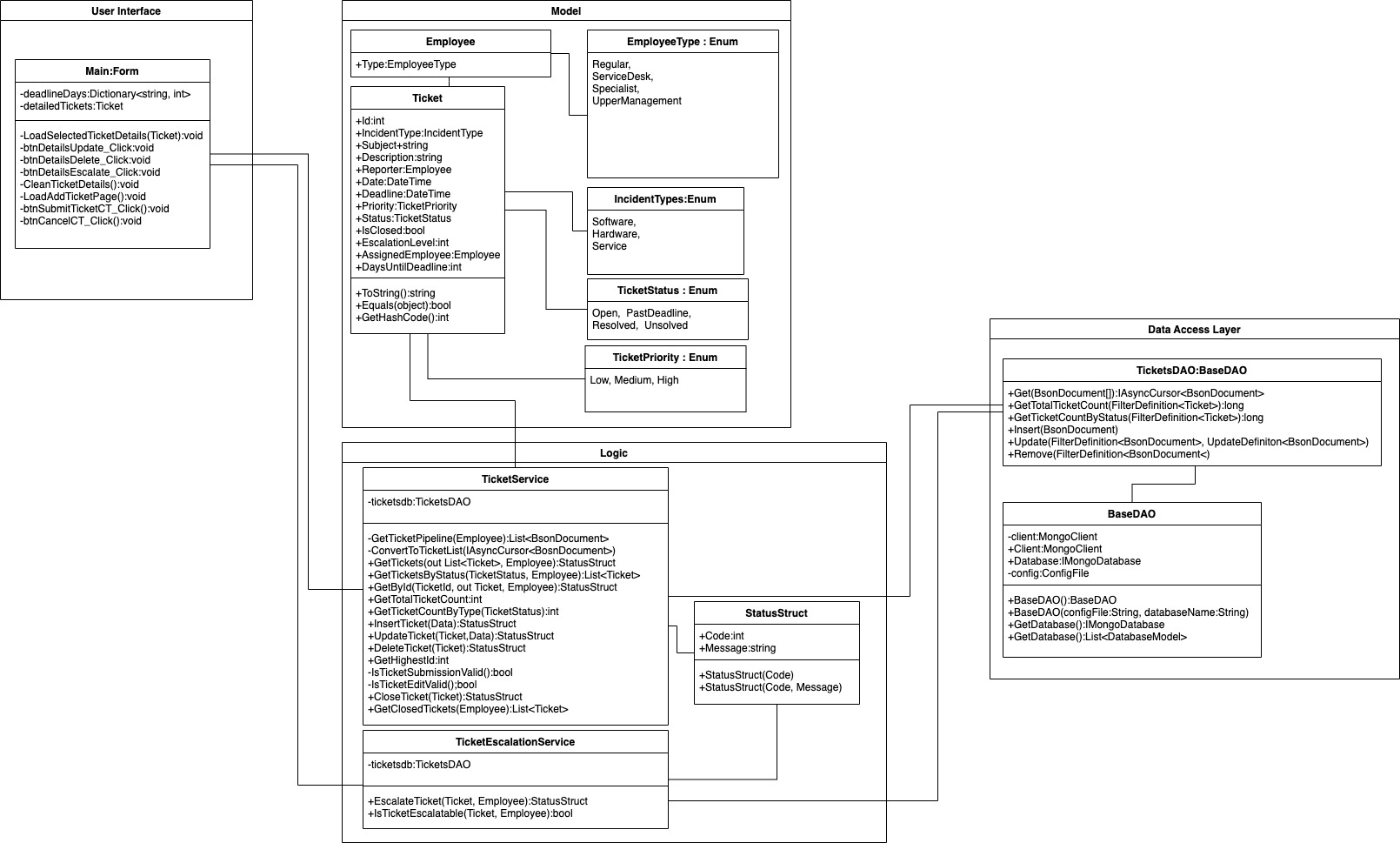
## 1.3.1. Adding employees and ticket transfer



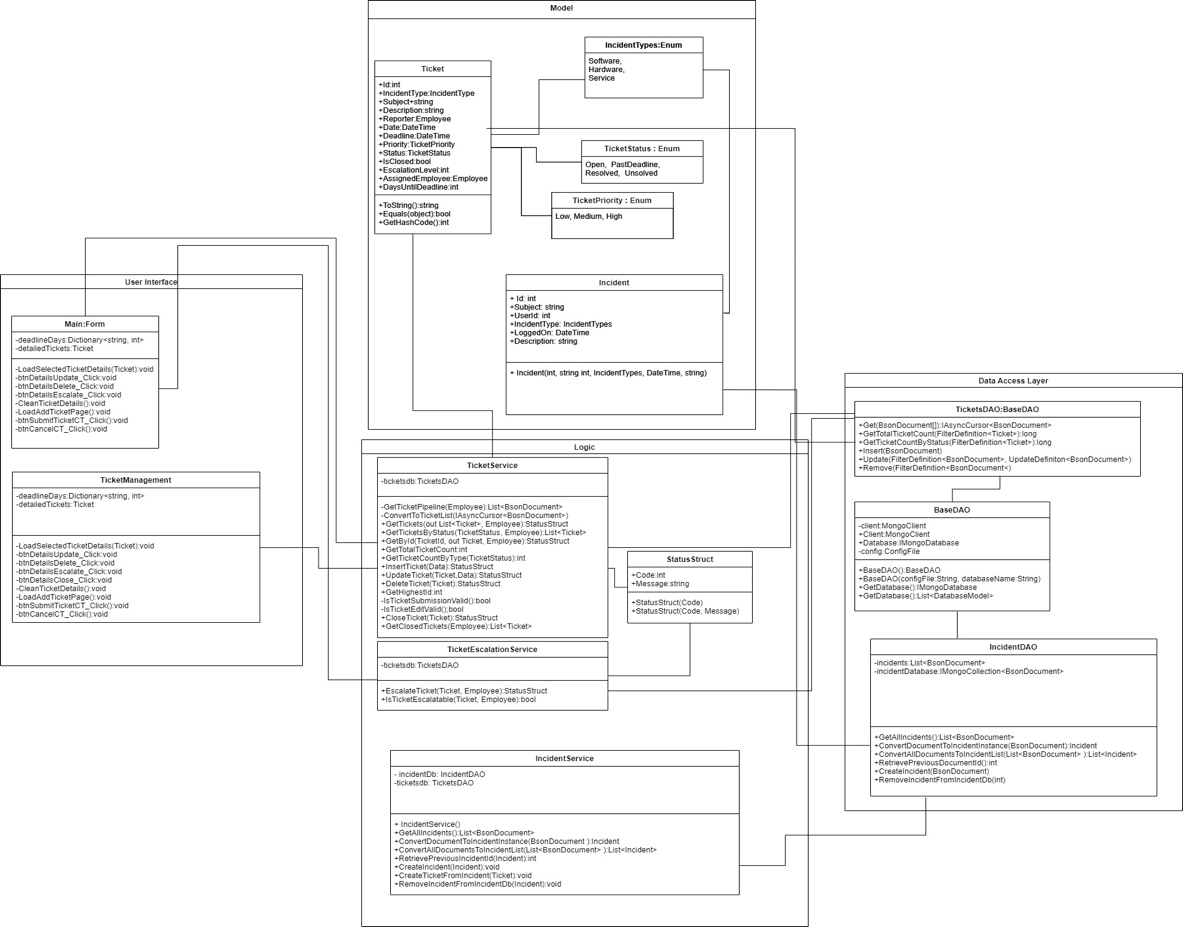
## 1.3.2. Dashboard and tickets and incident filtering



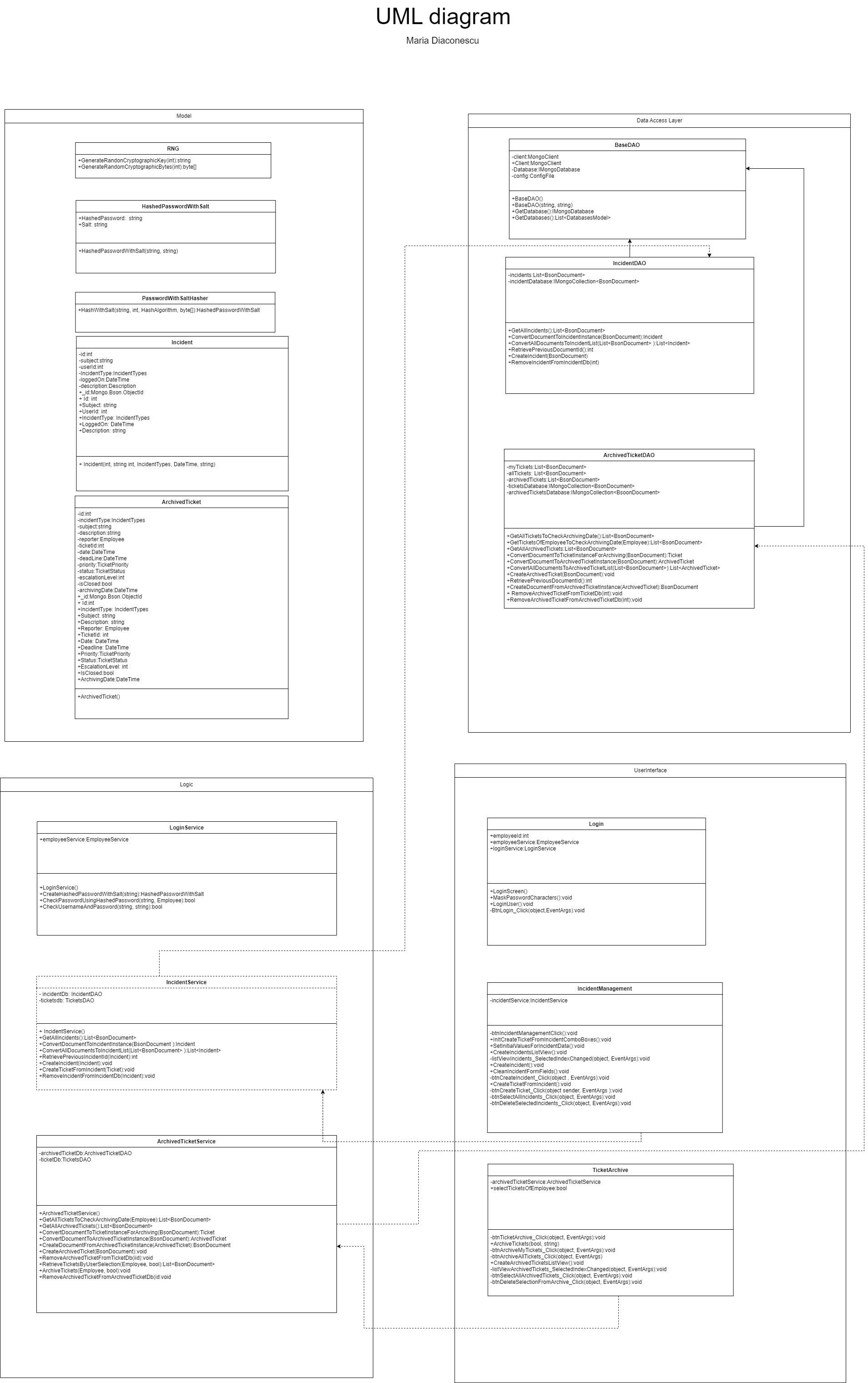
## 1.3.3. Create, read, update, delete tickets and escalate tickets



## 1.3.3. Display the tickets to the employee, close tickets and searching through tickets



## 1.3.4. Login, creating incidents and ticket archiving



# Konrad Figura - Design Choices

## Use of StatusStruct

Logic and use of „StatusStruct” as a return variable came out of the principle, where all error handling should be handled within the Logic layer. Nevertheless, if something goes wrong, I think it’s a good idea to let the user know that the thing they’ve tried to do went bad. StatusStruct contains the error code (where 0 means everything went OK), and a friendly message to the user. If a variable must be returned, it is passed as an “out” into the method’s constructor.

This design choice lets program handle the error in the Logic layer, making error handling independent from the Presentation layer. That way, presentation only handles what it’s supposed to do – presenting the data.

## TicketsDAO use of BsonDocument, instead of Ticket as a generic

TicketsDAO use of BsonDocument came from the larger flexibility use of BsonDocument. It seemed easier to directly insert, for example, just the ID of the Employee reporting the ticket, instead of having to implement a custom serialization/deserialization class.

It is also uncommon for documents to move between collections.

## Ticket Escalation as a field of the Ticket document

Originally, the idea was to have a separate collection called “EscalatedTickets”. This idea was abandoned, as in the situation, where there would be a need for ability of even higher escalation of the ticket, this would require creation of the new collection. Field can now be matched with the employee’s type, so as of right now: escalation level 0 = Help desk employee, escalation level 1 = specialist employee, escalation level 2 = upper management. Employees of matching escalation level can see those tickets.

## Use of Newtonsoft.Json

Newtonsoft.Json is used for importing the config.json file, which stores connection configuration to the MongoDB database.

JSON files are used, instead of App.config, simply because of preference choice, and easier deserialization.

## Inability to edit dates in submitted ticket

We’ve decided against allowing editing of the ticket date post submission, as it would allow employees to cheat on the deadlines. The only time it is allowed, is when converting incident into a ticket.

## Data transfer objects for InsertTicket and UpdateTicket

Data transfer objects have been implemented in InsertTicket and UpdateTicket, as the alternative would be having 9-10 parameters in both method’s constructors, making the code far less readable.

# Rodrigo Bange – Design Choices (Dashboard & Incident Filtering)

## Dashboard

The dashboard I designed has a navigation menu on the left side of the application with all the buttons to navigate around the application. The default page that gets opened in the application is the dashboard. On the dashboard there are four circular progression bars displayed. These circular progression bars may be clicked to navigate to the Ticket Management view and have the tickets loaded in of which the progression bar displays the amounts of. Upon loading, the dashboard calls to the Ticket Service where the request is converted through a Builder that creates the filter that gets sent to the Ticket DAO where the information gets requested from the database and send back to the DAO. The DAO sends it back to the Ticket DAO where the information gets returned as a “long” value which gets converted in the Ticket Service to Integer for the circular progression bars to be displayed. Do note that for both the ticket count and ticket searching the deadline value gets calculated by comparing the ticket’s deadline date and the current date.

## Long value to int conversion

MongoDB will return long values for count operations and long values are used for huge numbers and uses twice the memory compared to Int. So, I have decided that the long value gets converted to an Integer value in the Ticket Service to reduce memory usage.

Incident Ticket Filtering (Individual part)

For my choice of extra functionality, I have decided to do the incident ticket filtering. I have decided to use a textbox to allow the user to type multiple keywords of their own choice to filter the incidents by. The filtering is done by creating a filtered version of a pre-loaded incident list and displaying this filtered list in the list view. The filtering happens in the service layer (IncidentFilteringService) by grabbing the original pre-loaded list and the string of keywords. The string is then split by commas and spaces into an array of multiple strings. The incidents are then matched to the keywords by using exact matches, which means the keyword must exist within the incident’s values. If that is the case the incident is, then added to the filtered list and eventually the entire list is returned for display.

## Service Ticket Filtering

The Ticket Management has an overview of all the tickets and their corresponding information for the user to read and manage. My part was to do Incident Ticket Filtering, but I decided I might as well build the functionality for service tickets too. Which is why I created five buttons that the user can press that will call the Ticket Service to build a request by building a BsonDocument aggregation and sends this to the Ticket DAO where the request gets send to the database. The database then sends a list of values back that the DAO will send to the Ticket Service and eventually gets sent to the Main file for other processes to edit and for the graphical user interface to be displayed.

## Logic behind Ticket Count

The dashboard displays the number of all tickets that are currently registered in the database. These tickets are then returned as an Integer value for the dashboard circular progression bars to be displayed. The Ticket count is calculated by getting all the tickets that do NOT have an escalated value as these are raised up to the specialists instead of the service desk. Then the number of tickets is also calculated by the same rule however per Status category. The four statuses are “Open, Past Deadline, Unsolved and Solved”. These values are then displayed as mentioned before by the circular progression bars.

## Logic behind Ticket Filtering

On the Ticket Management page, I have created five buttons that perform a search on the database depending on the ticket status the user is trying to retrieve. The default search that will be performed when opening the page is to display all the tickets that are available to the service desk. Then the user can also press on the various buttons to search for a specific status of ticket such as “Open, Past Deadline, Unsolved and Solved” tickets. These tickets are then displayed in a view with all their corresponding information that the user might require.

## Circular Progression Bar

In the example given I saw a Circular Progression Bar and liked the idea, so I decided to look into it. Lots of people on forums recommended downloading this premade circular progression bar however it was quite costly in its performance and had way too many extra bits I wouldn’t need. So, I decided to build my own circular progression bar from scratch that would perform more efficiently and is simpler. It takes two values, the total and the amount of total and proceeds to calculate the percentage as well. Lastly it displays these two values (Both in number count and percentage) to the circular progression bar in the graphical user interface. The values displayed are taken from the database, if the database returns an exception, the circular progression bar instead will display “N/A” meaning “Not Available”. Do note that the database returns a long value for counting documents, this gets converted to integer in the Ticket Service and when displayed gets converted to float values so accurate percentages can be displayed.

## Faster Panel & Table Layout Panel

An issue I encountered while working with multiple panels and rendering was that the default panels don’t use Double Buffering and it’s not an element that can quickly be toggled on either. This can cause quite some performance issues when the window gets closed and reopened. You might see a couple seconds of nothing before object appear or that objects get rendered only partly and missing the other half. I wasn’t a fan of this and decided to look into figuring out a solution, so it didn’t seem like the application was falling apart with every single operation to the user. For this I had to create a new class that inherits from the standard provided Panel object and apply Double Buffering myself. It’s a quick and efficient way of solving an issue that shouldn’t have been there in the first place.

## Rounded Panels, buttons and custom icons and logos.

This is a design choice to make the application fit to the more modern design that is trending. Over the years applications have moved away from the square design and taken a more circular and minimalistic approach to designs. These, however, were not standard elements that you can adjust, and I had to program them myself to appear with rounded borders. The custom Icons and Logo are a styling choice to give the default WinForms a bit more of a touch. This adds onto the previous topic of rounded panels and buttons. In my opinion little touches like custom icons and logos can make an application look less boring and “default” as I would describe it. A little pop of life if I may call it like that.

# **Sara Eftekhar Azam – Design Choices**

## Searching

One of the requirements of the project was to add the ability to filter loaded tickets.

To implement this feature, I decided to first convert the written filtering query to lower case. I think that most people do not pay attention to the case of what they are writing in the search query and if the search is case sensitive, they might not find what they are looking for, so I decided to convert it to lower case.

This feature does not make a new query to the database and works on the loaded list of tickets that is already stored in the “allTickets” variable. If the search bar is empty, I show all of them. If it is not empty, I search for tickets that contain the written query in one of their major fields including status, description, reporter name, subject, or date. It is not the most sophisticated search, but I think it is a simple to understand search that users can understand easily, and it is easy to use and does what is needed from this feature.

After filtering the tickets by this condition, I display them.

## Close Ticket

At first, close tickets were distinguished in our system by setting their status to closed. However, after talking it over with the team, we decided that closed tickets are different from other tickets. Since we have both resolved and unresolved tickets in our system, having a closed ticket status seems redundant. So, we decided that closed tickets have different meanings that resolved or unresolved tickets. In our interpretation, we decided that closed tickets are the ones that have been closed so they cannot be changed or modified anymore.

Before making this change, the closed tickets were represented in our code by “closed” status. However, after this change, I added a new field to the ticket document named “IsClosed” which is set to “true” for closed tickets. This way, it is obvious that a closed ticket is different from a not closed ticket.

To show the user which tickets are closed and which tickets are not closed, I added a new column to the presentation of tickets. I thought about showing closed tickets by changing their color or font, but I decided that having a new column to show the closed ticket status is far more understandable by users. So, I added a column to show the closed state of the tickets.

## Close Ticket Search

One of the features of the application was the ability to search tickets by their status.

Before changing close tickets from status to a new field, searching database for closed tickets and then showing them was very similar to searching for unresolved tickets. However, after changing the meaning of closed tickets, the previous implementation for searching closed tickets would not work any longer, so I had to change it to be compatible with the new implementation.

I created a new method called “GetClosedTickets” in ticket service module. This method creates a query to database that gets all the tickets of the user with their “IsClosed” field set to true. After getting the data from the database, I show this data like other search conditions.

We have separated getting the data and displaying it in our code, so having a different condition only affects the part of getting the data and showing the data remains the same.

# **Fatmanur Vardar – Design Choices**

## Adding Employees

For adding employees, I created a registration form. In this form the service desk employee will be able to fill in all the information that is needed about the user. Since there are various employee types I used a drop-down list to select from.

To create a password for employees, I placed a button “create password”. This button first generates a random password for the user and displays it in the read-only textbox in the UI. Moreover, it creates the password’s hash and salt. Create password button disables after clicking to prevent unnecessary password generation.

To get ahead of registering a user with missing information, a warning message will be displayed if the service desk employee tries to register the user before filling in all the fields in the form and creating a password.

Finally, after registering the user, an empty register form will be loaded and with it the create password button will be enabled.

## Ticket Transfer

As my additional function, I chose the ticket transfer. This function enables transferring a ticket to another service desk employee. "Assigned employee" is a field of the Ticket document. This field represents the service desk employee who will be handling the ticket. The service desk employee who creates the ticket will be assigned to that ticket by default. With the Ticket Transfer function, the assigned employee will be changed easily.

In the ticket management tab, the service desk employee can select the ticket he wants to transfer from the list and choose the employee that the ticket will be transferred from the drop-down list. By clicking the “transfer ticket” button, the ticket will be transferred to the selected service desk employee.

We have decided to prevent transferring other service desk employees' tickets. Therefore, the application only allows transfer if the user matches the assigned employee of the selected ticket.

**Maria Diaconescu**

Adding incidents by regular employee

For this part of the application I created the classes Incident, IncidentDAO and IncidentService.

A regular employee can add incidents through a form, which is displayed to the user by the function DisplayDashboardForEmployee(). Also, based on the role of the user, the function DisplayTicketFormForEmployee displays the form for creating incidents by regular employees.

The fact that there are two separate entities, Incident and Ticked was based on the description of the application’s goal from the assignment. According to it, regular users can register service call