BIS Cozum

Banking Application

Author: Kaan Kahveci
Supervisor: Cengiz Uyar

Banking And Capital Markets
Date: August 9, 2023

Contents

Chapter 1

Introduction

This project encompasses the design and implementation of a desktop application targeting the management of key financial entities, namely Bills, Customers, and Transactions within the banking sector. The application, developed with Windows Presentation Foundation (WPF), operates through three distinct windows, each serving specific functionalities aimed at interacting with the database.

1.1 Technology Stack

Built on a robust technology stack, the application leverages the power of .NET Core and C#, alongside Microsoft SQL Server as the backend database. The following highlights the core technologies utilized in the development of the application:

- **.NET Core:** A cross-platform framework enabling the development of modern, high-performance applications.
- **C#:** The primary programming language employed, renowned for its object-oriented capabilities.
- **Microsoft SQL Server:** A relational database management system for storing and managing the application's data.
- **Windows Presentation Foundation (WPF):** A UI framework that creates desktop client applications, enabling a flexible design experience.

1.2 Functional Overview

The application presents three managing windows, each focused on the management of specific tables within the database:

1. **ManageBills:** Enables the fetching, insertion, and updating of billing information.
2. **ManageCustomers:** Manages customer data, allowing for detailed filtering and modification.
3. **ManageTransactions:** Handles transactional data, supporting various filtering options.

The application is designed with extensibility and scalability in mind, ensuring smooth operation and ease of future development.

1.3 Database Interaction

The interaction with the database is handled using ADO.NET, a data access technology from the Microsoft .NET Framework. It offers seamless connection capabilities with the SQL Server, providing high efficiency in executing SQL queries, including data fetching, insertion, and updating operations.

In summary, this project represents a comprehensive solution tailored to the needs of banking and financial management, integrating a solid technology stack, user-friendly interface, and a robust underlying database architecture.

Chapter 2

Functional Overview

2.1 Login Process

Upon launching the application, the user is presented with a *LoginWindow*. Here, the user must provide valid login credentials that are verified against the *Username* and *Password* columns in the *Users* table in the database. Successful login leads to the opening of the *MainWindow*.

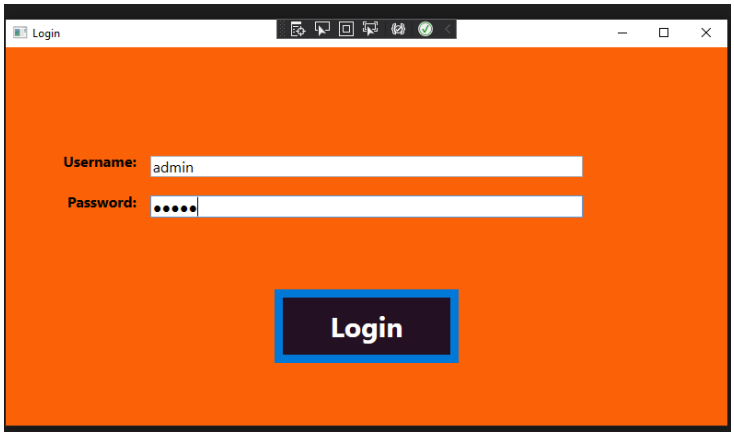


Figure 2.1: Login Window

2.1.1 MainWindow

MainWindow acts as the central hub of the application and consists of three buttons that allow the user to access the core functionalities:

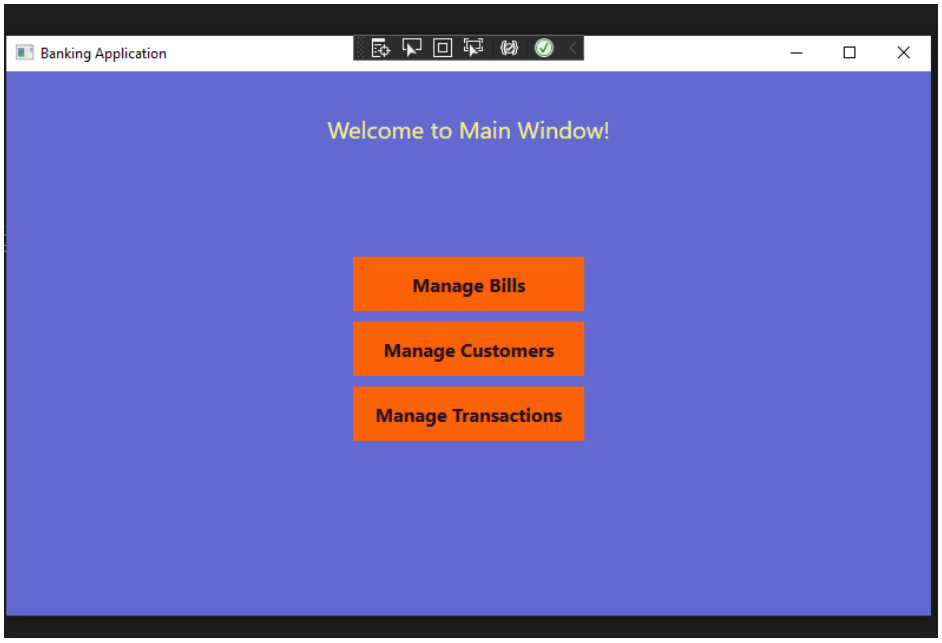


Figure 2.2: Main Window

- **Manage Bills:** Leads to *ManageBillsWindow*.
- **Manage Customers:** Leads to *ManageCustomersWindow*.
- **Manage Transactions:** Leads to *ManageTransactionsWindow*.

2.2 Managing Bills and Customers

ManageBillsWindow and *ManageCustomersWindow* have similar functionalities. They display respective details fetched from the database in a grid format. Each row in the grid has two actions: "update" and "delete". Additionally, filtering options are provided to narrow down the displayed details.

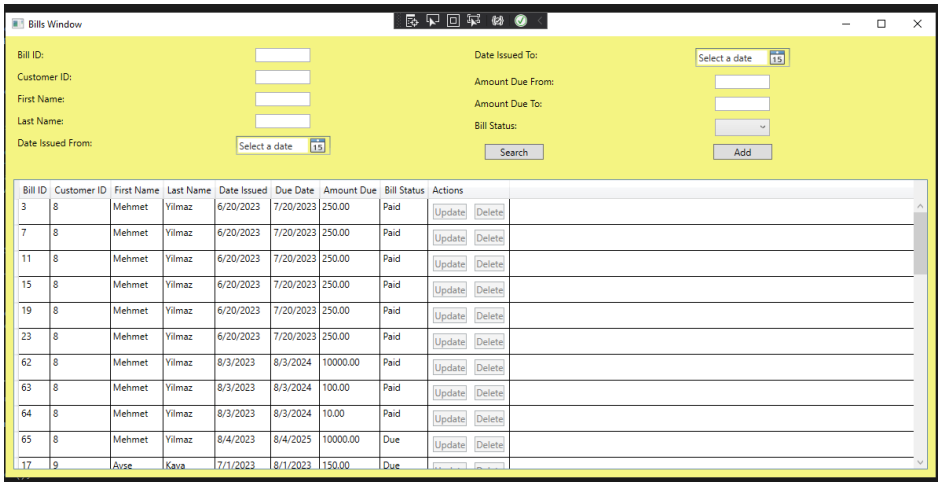


Figure 2.3: Manage Bills Window

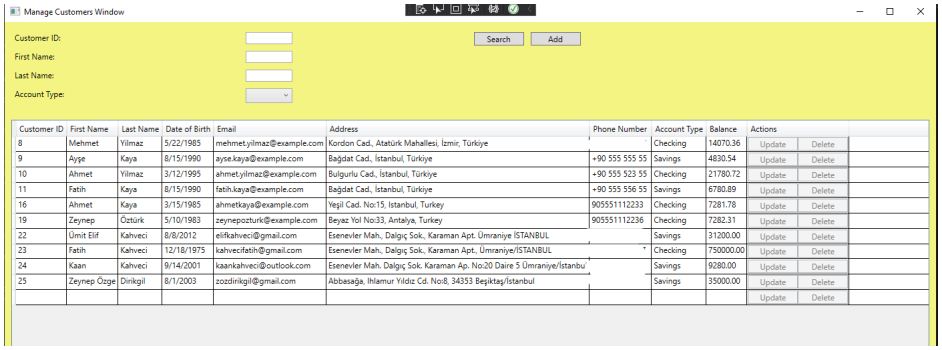


Figure 2.4: Manage Customers Window

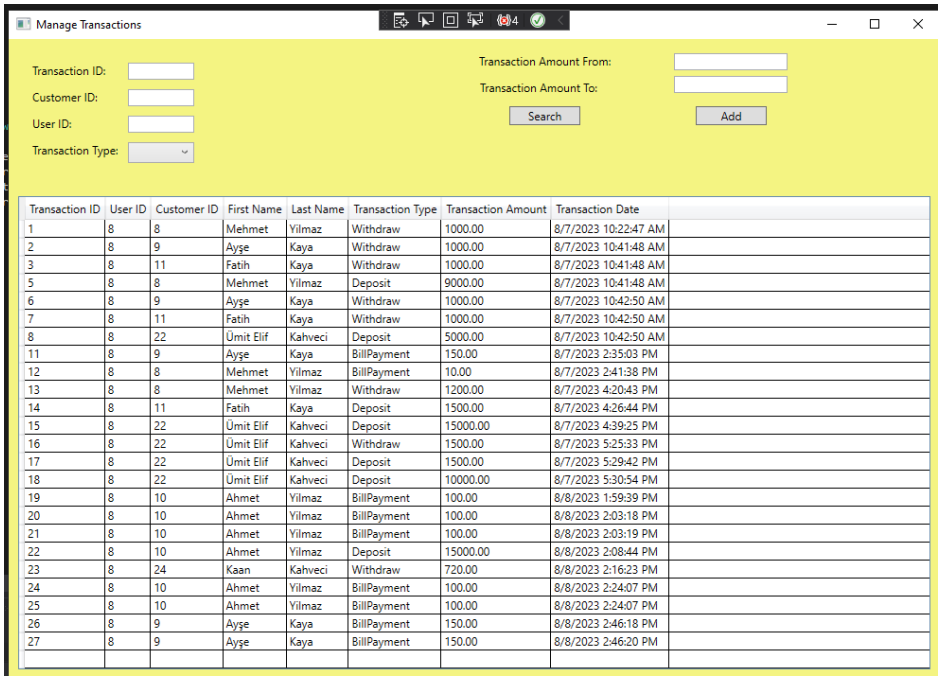
A button named "Add" opens a new window, either *AddCustomersWindow* or a similar window for adding bills. When updating existing customers or bills, this window appears with pre-filled fields corresponding to the selected item.

2.3 Managing Transactions

In *ManageTransactionsWindow*, the user can add new transactions with either $TransactionType = 1$ ("Withdraw") or $TransactionType = 2$ ("Deposit"). The system then updates the customer's "Balance" accordingly.

Unlike managing bills and customers, transactions cannot be deleted or updated manually. However, there is an automated process for handling $TransactionType = 3$ ("BillPayment"). When a bill's *BillStatus* changes from

1 ("Due") to 2 ("Paid"), the *ManageTransactionsViewModel* is notified. A new transaction is added with correct parameters, and the balance is adjusted through the *InsertTransaction* stored procedure in the database.



Transaction ID	User ID	Customer ID	First Name	Last Name	Transaction Type	Transaction Amount	Transaction Date
1	8	8	Mehmet	Yilmaz	Withdraw	1000.00	8/7/2023 10:22:47 AM
2	8	9	Ayşe	Kaya	Withdraw	1000.00	8/7/2023 10:41:48 AM
3	8	11	Fatih	Kaya	Withdraw	1000.00	8/7/2023 10:41:48 AM
5	8	8	Mehmet	Yilmaz	Deposit	9000.00	8/7/2023 10:41:48 AM
6	8	9	Ayşe	Kaya	Withdraw	1000.00	8/7/2023 10:42:50 AM
7	8	11	Fatih	Kaya	Withdraw	1000.00	8/7/2023 10:42:50 AM
8	8	22	Ümit Elif	Kahveci	Deposit	5000.00	8/7/2023 10:42:50 AM
11	8	9	Ayşe	Kaya	BillPayment	150.00	8/7/2023 2:35:03 PM
12	8	8	Mehmet	Yilmaz	BillPayment	10.00	8/7/2023 2:41:38 PM
13	8	8	Mehmet	Yilmaz	Withdraw	1200.00	8/7/2023 4:20:43 PM
14	8	11	Fatih	Kaya	Deposit	1500.00	8/7/2023 4:26:44 PM
15	8	22	Ümit Elif	Kahveci	Deposit	15000.00	8/7/2023 4:39:25 PM
16	8	22	Ümit Elif	Kahveci	Withdraw	1500.00	8/7/2023 5:25:33 PM
17	8	22	Ümit Elif	Kahveci	Deposit	1500.00	8/7/2023 5:29:42 PM
18	8	22	Ümit Elif	Kahveci	Deposit	10000.00	8/7/2023 5:30:54 PM
19	8	10	Ahmet	Yilmaz	BillPayment	100.00	8/8/2023 1:59:39 PM
20	8	10	Ahmet	Yilmaz	BillPayment	100.00	8/8/2023 2:03:18 PM
21	8	10	Ahmet	Yilmaz	BillPayment	100.00	8/8/2023 2:03:19 PM
22	8	10	Ahmet	Yilmaz	Deposit	15000.00	8/8/2023 2:08:44 PM
23	8	24	Kaan	Kahveci	Withdraw	720.00	8/8/2023 2:16:23 PM
24	8	10	Ahmet	Yilmaz	BillPayment	100.00	8/8/2023 2:24:07 PM
25	8	10	Ahmet	Yilmaz	BillPayment	100.00	8/8/2023 2:24:07 PM
26	8	9	Ayşe	Kaya	BillPayment	150.00	8/8/2023 2:46:18 PM
27	8	9	Ayşe	Kaya	BillPayment	150.00	8/8/2023 2:46:20 PM

Figure 2.5: Manage Transactions Window

2.4 Conclusion

The application integrates a broad spectrum of functionalities that allow for detailed management of banking entities. Through the synergy of various windows and logical structures, users are granted an intuitive interface to interact with the underlying data, efficiently managing bills, customers, and transactions. Careful attention to transactional integrity and automated processes further contribute to the application's robustness and utility in the banking domain.

Chapter 3

Technical Overview

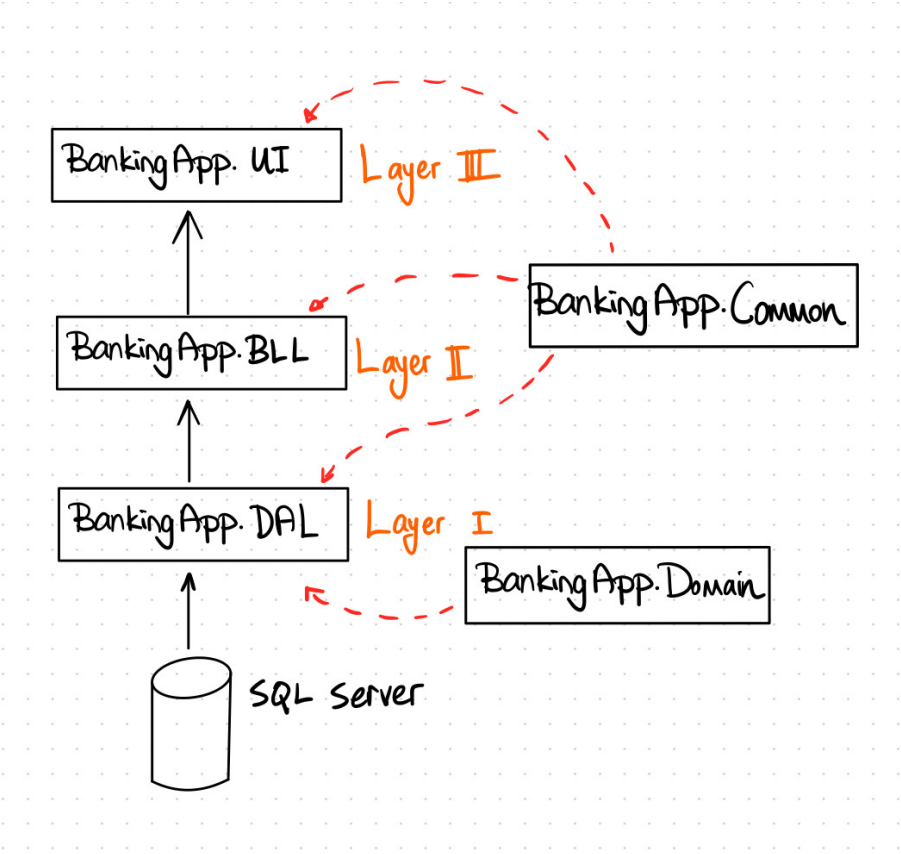


Figure 3.1: General Overview of the Architecture

3.1 Architecture of the Project

The architecture of the BankingApp project is modular and layered, designed for maintainability and scalability. It consists of five class libraries:

3.1.1 BankingApp.Domain

This library serves as the domain layer of the application. It includes:

- Entity classes representing the core business objects within the system.
- Detail classes for displaying records in the user interface.
- Filter classes for querying specific fields or criteria.

In the Domain layer, we define the core entities that represent the main constructs within the banking application. Here, we are focusing on the Bill-related classes for explanation sake:

Bill.cs, BillDetails.cs, BillFilter.cs as example

```
using System;

namespace BankingApp.Domain
{
  // The Bill class represents the basic attributes of a
    bill.
  public class Bill
  {
    public int BillId { get; set; }
    public int CustomerId { get; set; }
    public DateTime DateIssued { get; set; }
    public DateTime DueDate { get; set; }
    public decimal AmountDue { get; set; }
    public int BillStatus { get; set; }
  }

  // The BillDetails class extends the basic Bill
   attributes with additional details.
  public class BillDetails
  {
    // properties
    public Bill ToBill()
    {
```

```
      return new Bill
      {
        // property assignments
      };
    }
  }

  // The BillFilter class is used to filter bills based
   on various criteria.
  public class BillFilter
  {
    // properties
  }
}
```

3.1.2 BankingApp.DAL

Acting as Layer 1 (Data Access Layer), this library is responsible for interacting with the database and stored procedures. It contains classes like `BillData.cs`, each with implementations corresponding to specific stored procedures within the database.

BillData.cs example

Here's an example of how a bill is updated in the database:

```
public bool UpdateBill(Bill bill)
{
  using (SqlConnection connection = new SqlConnection(
   this.connectionString))
  {
    using (SqlCommand command = new SqlCommand("dbo.
   UpdateBill", connection))
    {
      // Providing necessary parameters to SP
      // Command settings
      connection.Open();
      command.ExecuteNonQuery();

      // Error handling
      return true;
    }
  }
```

```
}
```

Database Connection

The BankingApp.DAL layer utilizes a connection string to interact with the database. This connection string is stored in a configuration file named `App.config`, providing a centralized and secure location for database connection information.

```
private readonly string connectionString = ConfigurationManager.ConnectionStrings["BankingAppConnectionString"].ConnectionString;
```

Figure 3.2: Statement used in every DAL class

By managing the connection string in this manner, the application benefits from added flexibility and maintainability, as changes to the database connection can be made without altering the code.

3.1.3 BankingApp.BLL

This is Layer 2 (Business Logic Layer), acting as an intermediary between the DAL and the UI layers. It includes service classes, such as `BillService.cs`, for managing the business rules and operations related to different entities.

BillService.cs as example

```
public bool UpdateBill(Bill bill)
{
  try
  {
    // Check if customer exists and update the bill
  }
  catch(DatabaseException ex)
  {
    throw new BusinessException("Failed␣to␣update␣bill:␣
   " + ex.Message);
  }
}
```

3.1.4 BankingApp.UI

This library focuses on user interface-related tasks and adheres to the MVVM design pattern. Within this library, various subfolders organize components as follows:

- **Commands**: Command implementations for user actions.
- **Events**: Event classes for handling application-wide events.
- **NavigationServices**: Classes for managing navigation between views.
- **ViewModels**: View models that handle UI logic.
- **Views**: User interface views or windows.

The library also leverages dependency injection and the Observer Pattern through event aggregators.

Dependency Injection in UI Layer

Dependency injection is employed in the BankingApp.UI layer to ensure that the necessary services are readily available for the view models. This approach facilitates a more modular and maintainable codebase by decoupling the dependencies between components.

In the `App.xaml.cs` file, the services required by the application are instantiated and injected into the `LoginViewModel`. Subsequently, `LoginViewModel` injects the necessary services into other view models.

```
using System.Windows;
using BankingApp.BLL;
using BankingApp.DAL;
using BankingApp.UI.NavigationServices;
using BankingApp.UI.ViewModels;
using BankingApp.Common.Events;

namespace BankingApp.UI
{
  public partial class App : Application
  {
    protected override void OnStartup(StartupEventArgs e
   )
    {
      base.OnStartup(e);
```

```
        UserService userService = new UserService(new
    UserData());
        CustomerService customerService = new
    CustomerService(new CustomerData());
        BillService billService = new BillService(new
    BillData(), customerService);
        ParameterService parameterService = new
    ParameterService(new ParameterData());
        TransactionService transactionService = new
    TransactionService(new TransactionData());

        IEventAggregator eventAggregator = new
    EventAggregator();

        TransactionHandler transactionHandler = new
    TransactionHandler(eventAggregator,
    transactionService);

        INavigationService navigationService = new
    NavigationService();

        LoginViewModel loginViewModel = new LoginViewModel
    (userService, customerService, billService,
    parameterService, navigationService, eventAggregator,
     transactionService);

        navigationService.Navigate(loginViewModel);
      }
    }
}
```

Listing 3.1: App.xaml.cs

SaveChanges method from the AddBillsViewModel.cs as example

Below is a method that is responsible for saving changes to a bill:

```
public void SaveChanges(object parameter)
{
  try
  {
    // Bill handling logic
  }
  catch (BusinessException ex)
```

```
  {
    MessageBox.Show(ex.Message, "Error",
   MessageBoxButton.OK, MessageBoxImage.Error);
  }
  catch (Exception ex)
  {
    MessageBox.Show("An␣unexpected␣error␣occurred.␣
   Please␣try␣again.", "Error", MessageBoxButton.OK,
   MessageBoxImage.Error);
  }
}
```

This architecture ensures that each view model has access to the services it needs while maintaining separation of concerns. Dependency injection allows for more straightforward testing, as dependencies can be mocked or replaced easily, enhancing the overall maintainability and robustness of the application.

This design allows each component to focus on its specific responsibilities without needing to know the details of its dependencies, fostering a more maintainable and flexible codebase.

3.1.5 BankingApp.Common

This additional library contains classes that are used across multiple layers of the application. It includes:

- `IEventAggregator.cs` and `EventAggregator.cs` for implementing the Observer Pattern.
- `UserSession.cs` for tracking the logged-in user and associated UserID.

The structure of these class libraries ensures a clean separation of concerns and promotes modularity, maintainability, and scalability within the BankingApp system.

Event Aggregator Pattern

The Event Aggregator Pattern is used to manage events across different components of the application. It allows objects to communicate with each other without knowing each other's identity.

```
namespace BankingApp.Common.Events
{
```

```
  public interface IEventAggregator
  {
    void Publish<TEvent>(TEvent eventToPublish);
    void Subscribe<TEvent>(Action<TEvent> action);
  }

  public class EventAggregator : IEventAggregator
  {
    // Implementation details
  }
}
```

The 'IEventAggregator' interface defines two primary methods: 'Publish' for broadcasting an event, and 'Subscribe' for subscribing to an event. The 'EventAggregator' class is responsible for storing and invoking the subscribed actions for the events.

User Session

The 'UserSession' class is used to keep track of the currently logged-in user, which enables the system to associate actions with a specific UserID.

```
using BankingApp.Domain;

namespace BankingApp.Common
{
  public static class UserSession
  {
    public static User CurrentUser { get; set; }
  }
}
```

This simple class leverages the static property 'CurrentUser' to allow global access to the user's information while ensuring that only one user is active at a time.

These common utilities play a vital role in coordinating actions and maintaining state across different layers, thereby enhancing the overall cohesiveness and functionality of the BankingApp system.

3.2 Database Schema and Diagram

3.2.1 Overview

The database for the BankingApp project consists of five main tables and a set of stored procedures to manage CRUD operations. These tables manage users, customers, bills, transactions, and a parameter system to handle user interface-friendly descriptions and corresponding database codes.

3.2.2 Tables

- **Users**: Manages user information.
  - *UserId* (PK): Unique identifier for the user.
  - *Username*: The username for login.
  - *Password*: The password for login.
- **Customers**: Manages customer information.
  - *CustomerId* (PK): Unique identifier for the customer.
  - *FirstName*
  - *LastName*
  - *DateOfBirth*
  - *Email*
  - *Address*
  - *PhoneNumber*
  - *AccountType*
  - *Balance*
- **Bills**: Manages bill information.
  - *BillId* (PK): Unique identifier for the bill.
  - *CustomerId* (FK)
  - *DateIssued*
  - *DueDate*
  - *AmountDue*
  - *BillStatus*

- **Transactions**: Manages transaction information.
  - *TransactionId* (PK): Unique identifier for the transaction.
  - *UserId* (FK) : Represents the user (banker) that have made the transaction
  - *TransactionAmount*
  - *TransactionType*
  - *TransactionDate*
  - *CustomerId* (FK) : Represents the customer who is related to the bill
  - *PhoneNumber*
  - *AccountType*
  - *Balance*
- **Parameters**: A special table to map INT codes to UI-friendly descriptions. Used for attributes like BillStatus, TransactionType, and AccountType.
  - *Type* (PK): Type of parameter.
  - *Code* (PK): Code representing the value.
  - *Description*: UI-friendly description.
  - *Active*: Boolean to indicate whether the parameter is active.

3.2.3 Stored Procedure Example: `dbo.UpdateBill`

The `dbo.UpdateBill` stored procedure showcases the typical structure of the stored procedures. It handles the update of a bill, and when the BillStatus changes to "Paid", it adds a corresponding transaction and updates the customer's balance.

```
USE dbproje;

GO

DROP PROCEDURE IF EXISTS dbo.UpdateBill;

GO

CREATE PROCEDURE dbo.UpdateBill
```

```
  @UserId INT, -- Add UserId parameter
  @BillId INT,
  @CustomerId INT,
  @DateIssued DATE,
  @DueDate DATE,
  @AmountDue DECIMAL(18,2),
  @BillStatus INT,
  @ErrorMessage NVARCHAR(MAX) OUTPUT -- Add an output
     parameter for the error message
AS
BEGIN;
  BEGIN TRY;
  BEGIN TRANSACTION;

    -- Check if the BillStatus is being updated to "Paid
       "
    IF @BillStatus = 2
    BEGIN
      DECLARE @TransactionId INT;
      DECLARE @InsertTransactionError NVARCHAR(MAX);

      -- Call the InsertTransaction procedure to handle
         the payment
      EXEC dbo.InsertTransaction
        @UserId = @UserId, -- Provide UserId from the
           parameter
        @CustomerId = @CustomerId,
        @TransactionAmount = @AmountDue,
        @TransactionType = 3, -- 3: BillPayment
        @NewTransactionId = @TransactionId OUTPUT,
        @ErrorMessage = @InsertTransactionError OUTPUT;

      -- If there was an error inserting the transaction
         , throw an exception
      IF @InsertTransactionError IS NOT NULL
      BEGIN
        SET @ErrorMessage = @InsertTransactionError;
        THROW 50000, @InsertTransactionError, 1;
      END
    END

    -- Continue updating the bill as usual
    UPDATE Bills
```

```
    SET CustomerId = @CustomerId,
      DateIssued = @DateIssued,
      DueDate = @DueDate,
      AmountDue = @AmountDue,
      BillStatus = @BillStatus
    WHERE BillId = @BillId;

  COMMIT TRANSACTION;
  END TRY
  BEGIN CATCH;
    IF(@@TRANCOUNT > 0)
    BEGIN;
      ROLLBACK TRANSACTION;
    END;
    SET @ErrorMessage = ERROR_MESSAGE();
    RETURN -1;
  END CATCH;
  RETURN 0;
END;

END;
```

Listing 3.2: Stored Procedure dbo.UpdateBill

This procedure calls another procedure, `dbo.InsertTransaction`, and performs a conditional update with error handling. It demonstrates the use of transactions and rollbacks to ensure data integrity and atomic changes.

3.2.4 Transactions and Error Handling

Most of the stored procedures follow a similar structure of using transactions and rollbacks to ensure data integrity. They make changes atomically and include comprehensive error handling to gracefully manage exceptions.

3.2.5 Database Diagram

This diagram visualizes the relationships between the tables and provides a concise overview of the database design.

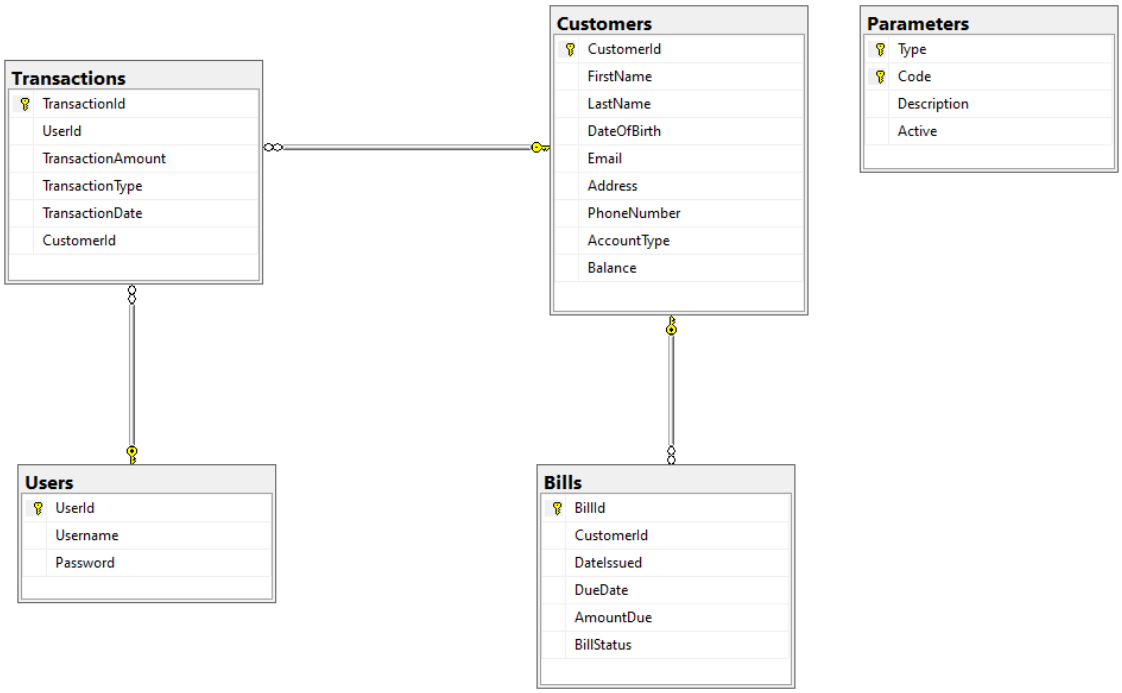


Figure 3.3: Database Schema Diagram for BankingApp

3.2.6 Conclusion

The database design effectively supports the BankingApp's requirements by maintaining a clear structure and following best practices in stored procedures, transactions, error handling, and user-friendly mapping of codes to descriptions.

3.3 Key Code Snippets and Insights from the Application

This section will provide a detailed explanation of some key code components used within the application.

3.3.1 EventAggregator.cs

The `EventAggregator` class is a central part of the application's event-driven architecture. It allows different parts of the application to publish and subscribe to events.

```
using System;
using System.Collections.Generic;

namespace BankingApp.Common.Events
{
  public class EventAggregator : IEventAggregator
  {
    private readonly Dictionary<Type, List<Action<object
   >>> _eventSubscribers = new Dictionary<Type, List<
   Action<object>>>();

    public void Publish<TEvent>(TEvent eventToPublish)
    {
      var eventType = typeof(TEvent);
      if (_eventSubscribers.ContainsKey(eventType))
      {
        var eventActions = _eventSubscribers[eventType];
        foreach(var eventAction in eventActions)
        {
          eventAction(eventToPublish);
        }
      }
    }
    public void Subscribe<TEvent>(Action<TEvent> action)
    {
      var eventType = typeof(TEvent);
      if (_eventSubscribers.ContainsKey(eventType))
      {
        _eventSubscribers[eventType].Add(obj => action((
   TEvent)obj));
      }
      else
      {
        _eventSubscribers[eventType] = new List<Action<
   object>> { obj => action((TEvent)obj) };
      }
    }
  }
```

```
}
```

Listing 3.3: EventAggregator.cs

The `EventAggregator` maintains a dictionary of event subscribers. The `Subscribe` method allows other classes to register actions to take when a specific event is published. The `Publish` method triggers all actions registered for a particular event type.

3.3.2 RelayCommand.cs

The `RelayCommand` class implements the `ICommand` interface, allowing the binding of commands in MVVM pattern.

```
using System;
using System.Windows.Input;

namespace BankingApp.UI.Commands
{
  public class RelayCommand : ICommand
  {
    private readonly Action<object> _execute;
    private readonly Predicate<object> _canExecute;

    public RelayCommand(Action <object> execute,
   Predicate <object> canExecute)
    {
      this._execute = execute;
      this._canExecute = canExecute;
    }

    public event EventHandler CanExecuteChanged
    {
      add { CommandManager.RequerySuggested += value; }
      remove { CommandManager.RequerySuggested -= value;
    }
    }

    public bool CanExecute(object parameter)
    {
      return _canExecute == null || _canExecute(
   parameter);
    }
```

```
    public void Execute(object parameter)
    {
      _execute(parameter);
    }
  }
}
```

Listing 3.4: RelayCommand.cs

`RelayCommand` provides a straightforward way to link actions with user interface commands, including conditions under which they can be executed, handled by the `CanExecute` method.

3.3.3 BillsViewModel and Its Usage

The `BillsViewModel` class exemplifies how `EventAggregator` and `RelayCommand` work together to create a responsive, event-driven user interface.

```
using BankingApp.BLL;
using BankingApp.Domain;
using BankingApp.UI.Commands;
using BankingApp.UI.NavigationServices;
using System.Collections.ObjectModel;
using BankingApp.UI.Events;
using BankingApp.Common.Events;

namespace BankingApp.UI.ViewModels
{
  public class BillsViewModel : BaseViewModel
  {
    private readonly BillService _billService;
    private readonly CustomerService _customerService;
    private readonly ParameterService _parameterService;
    private readonly INavigationService
   _navigationService;
    private readonly IEventAggregator _eventAggregator;

    private BillDetails _selectedBill;
    public BillDetails SelectedBill
    {
      get { return _selectedBill; }
      set
      {
        _selectedBill = value;
```

```
      OnPropertyChanged();
    }
  }

  private BillFilter _billFilter;
  public BillFilter BillFilter
  {
    get { return _billFilter; }
    set
    {
      _billFilter = value;
      OnPropertyChanged();
    }
  }

  public RelayCommand SearchBillsCommand { get; set; }

  public ObservableCollection<BillDetails> Bills { get
; set; }

  public ObservableCollection<Parameter> BillStatuses
{ get; set; }

  public RelayCommand DeleteBillCommand { get; set; }
  public RelayCommand UpdateBillCommand { get; set; }
  public RelayCommand NavigateToAddBillCommand { get;
set; }

  public BillsViewModel(BillService billService,
CustomerService customerService, ParameterService
parameterService, INavigationService
navigationService, IEventAggregator eventAggregator)
  {
    _billService = billService;
    _customerService = customerService;
    _parameterService = parameterService;
    _navigationService = navigationService;
    _eventAggregator = eventAggregator;
    _eventAggregator.Subscribe<BillUpdatedEvent>(
OnBillUpdated);

    Bills = new ObservableCollection<BillDetails>(
_billService.FetchAllBillDetails());
```

```
    BillFilter = new BillFilter();

    BillStatuses = new ObservableCollection<Parameter
>(_parameterService.FetchParametersByType("BillStatus
"));

    DeleteBillCommand = new RelayCommand(DeleteBill, _
 => SelectedBill != null);
    UpdateBillCommand = new RelayCommand(UpdateBill, _
 => SelectedBill != null);
    NavigateToAddBillCommand = new RelayCommand(
NavigateToAddBill, _ => true);
    SearchBillsCommand = new RelayCommand(SearchBills,
 _ => true);
  }

  private void DeleteBill(object obj)
  {
    _billService.DeleteBill(SelectedBill.BillId);
    Bills.Remove(SelectedBill);
  }

  private void UpdateBill(object obj)
  {
    var addBillsViewModel = new AddBillsViewModel(
_billService, _customerService, _parameterService,
_navigationService, _eventAggregator, SelectedBill.
ToBill());
    _navigationService.Navigate(addBillsViewModel);
  }

  private void NavigateToAddBill(object obj)
  {
    var addBillsViewModel = new AddBillsViewModel(
_billService, _customerService, _parameterService,
_navigationService, _eventAggregator);
    _navigationService.Navigate(addBillsViewModel);
  }

  private void OnBillUpdated(BillUpdatedEvent
billUpdatedEvent)
  {
```

```
        //Refresh the BillsCollection
        Bills = new ObservableCollection<BillDetails>(
    _billService.FetchAllBillDetails());

        // Notify any data bindings that the Bills
    property has changed
        OnPropertyChanged(nameof(Bills));
      }

      private void SearchBills(object obj)
      {
        var bills = _billService.SearchBillDetails(
    BillFilter);
        Bills.Clear();

        foreach (var bill in bills)
        {
          Bills.Add(bill);
        }

        OnPropertyChanged(nameof(Bills)); // Notify UI
    about the changes
      }
    }
}
```

Listing 3.5: BillsViewModel.cs

Explanation of Key Components:

- **RelayCommand** is used to bind UI commands to methods in the ViewModel.
- **EventAggregator** facilitates communication between different parts of the application. For instance, it enables the ViewModel to subscribe to the `BillUpdatedEvent`, allowing it to refresh the Bills collection when a bill is updated.
- **Actions, Events, and Delegates**: The combination of actions, events, and delegates in the code enables a highly flexible and responsive design. It allows the application to respond to changes in one part of the system from another part without tight coupling.