Словесное описание алгоритма работы программы

Программа для служащих гостиницы архитектурно разделена на три части: модель данных, расположенную в проекте **Model**, модуль связи с базой данных SQL-сервера, расположенный в проекте **Database** и интерфейс пользователя – в проекте **Reception**.

Модель данных должна быть независима от интерфейса и от средств хранения данных, поскольку интерфейс может быть разным, не только на основе технологии **WinForms**, а сохранение данных модели может производиться и в локальный файл, и в базу данных, и в «облако».

Основной класс, хранящий все другие коллекции сущностей и предоставляющий их другим частям программы – интерфейсу пользователя (классам из **Reception**) и модулю связи с базой данных (классам из **Database**), носит имя **Hotel**.

**public** **class** Hotel

{

**public** Categories Categories { **get**; **set**; }*// категории*

**public** Services Services { **get**; **set**; } *// услуги*

**public** Rooms Rooms { **get**; **set**; } *// номера*

**public** EmployeeRoles EmployeeRoles { **get**; **set**; } *// должности сотрудников*

**public** RegistryStaff RegistryStaff { **get**; **set**; } *// сотрудники*

**public** Clients Clients { **get**; **set**; } = new Clients(); *// клиенты*

**public** Reservations Reservations { **get**; **set**; } *// бронирование*

**public** Transfers Transfers { **get**; **set**; } *// трансферы*

**public** PayChannels PayChannels { **get**; **set**; } *// платные каналы*

**public** AccordancePayChannels AccordancePayChannels { **get**; **set**; } *// подписки на каналы*

**public** Employee CurrentUser { **get**; **set**; } *// ссылка на текущего пользователя программы*

}

Фрагмент класса Hotel

Вышеуказанные коллекции объектов представляют собой списки сущностей, представляющих модель работы гостиницы:

* **Rooms** – содержит объекты **Room**, представляющие номера гостиницы.
* **Reservations** – содержит объекты **Reservation**, представляющие записи в журнале «бронирования» - резервирования номеров на указанный диапазон дат за указанным клиентом.
* **Clients** – содержит объекты **Client** с информацией о клиентах гостиницы.
* **Transfers** – содержит заказы на услугу доставки клиентов из указанных транспортных узлов (вокзалов, аэропортов) в указанное время и количество мест, к подъезду гостиницы.
* **RegistryStaff** – содержит объекты **Employee** с данными о сотрудниках гостиницы.
* **AccordancePayChannels** – содержит сведения о подписке клиентов на подключаемые платные кабельные каналы телевидения.
* **Categories** – справочник, содержит перечень наименований, обозначающих степень комфортности номера. Данные этого справочника используются при настройке объектов номеров **Room**.
* **Services** – справочник, содержит перечень наименований и стоимости услуг, предоставляемых в номер по выбору администратора. Данные этого справочника используются при настройке объектов номеров **Room**.
* **EmployeeRoles** – справочник, содержит перечень наименований должностей сотрудников, с указанием должностного оклада и привилегий при работе с программой. Данные этого справочника используются при настройке объектов сотрудников Employee.
* **PayChannels** – справочник, содержит перечень наименований платных каналов кабельного телевидения, подключаемых при оформлении подписки. Данные этого справочника используются при настройке объектов подписок **AccordancePayChannel**.

Всего десять сущностей представлено в этом списке, далее рассмотрим работу каждой сущности более подробно.

При начальной загрузке программы, при инициализации основной формы **MainForm**, создаётся пустой объект модели **Hotel** и привязывается к локальной переменной **\_hotel**.

private Hotel \_hotel = new Hotel(); // модель

В конструкторе формы **MainForm** изначально запрещается пункт главного меню «Вход…», пока данные не будут загружены в модель. Также вызывается метод **BuildData()**, который может использоваться для загрузки тестовых данных в модель, но в окончательной версии не используется.

Загрузка данных в модель происходит в обработчике первоначальной загрузки формы **MainForm\_Load()**, при вызове метода формы **LoadFromBaseAsync()**:

/// <summary>

/// Загрузка данных из базы асинхронно

/// </summary>

private void LoadFromBaseAsync()

{

tsmiLogin.Enabled = false;

Task.Run(() =>

{

\_hotel = SaverLoader.LoadFromBase(Properties.Settings.Default.ConnectionString);

var method = new MethodInvoker(() =>

{

tsmiLogin.Enabled = true;

var result = SaverLoader.OperationResult;

tsslStatusLabel.Text = string.IsNullOrWhiteSpace(result)

? "Готово" : result.Substring(0, result.IndexOf('.') + 1);

statusStrip1.Refresh();

});

if (InvokeRequired)

BeginInvoke(method);

else

method();

});

}

Данные загружаются при вызове метода **LoadFromBase()** статического класса **SaverLoader**, отвечающего в модели за сохранение и загрузку данных модели для внешних носителей данных. Параметром метода **LoadFromBase()** выступает строка соединения с базой данных, которая хранится в настройках приложения **Properties.Settings.Default.ConnectionString**, а результатом работы метода – ссылка на вновь созданный объект класса **Hotel**, наполненный данными.

/// <summary>

/// Метод загрузки сохранённой ранее конфигурации из базы данных

/// </summary>

/// <param name="connection"></param>

/// <returns></returns>

public static Hotel LoadFromBase(string connection)

{

var hotel = new Hotel();

var server = new Database.SqlServer { Connection = connection };

// категории

var dataSet = server.GetRows("Categories");

if (dataSet.Tables.Count > 0)

foreach (var row in dataSet.Tables[0].Rows.Cast<DataRow>())

{

if (row.ItemArray.Length != 2) continue;

hotel.Categories.Add(Guid.Parse(row.ItemArray[0].ToString()),

row.ItemArray[1].ToString());

}

OperationResult = server.LastError;

// услуги

dataSet = server.GetRows("Services");

// и далее по тексту...

}

Вначале выполнения метода **LoadFromBase()** создается пустой объект **hotel**, потом создается объект взаимодействия с SQL-сервером **SqlServer**, описание класса которого находится в проекте **Database**, и которому передаётся строка подключения из параметра метода. Объект **server** выполняет свой метод **GetRows()**, параметром которого является наименование таблицы из базы данных, а результатом работы – набор данных системного типа **DataSet**.

При нормальной работе возвращается набор данных с одной таблицей, содержимое которой мы просматриваем в цикле **foreach**, контролируя, что каждая считываемая запись имеет два значения (*два значения именно для таблицы «Categories», у которой определено два столбца: IdCategory и NameCategory, а для других таблиц число значений может быть иное*). Полученные значения присваиваются новой записи в коллекции объектов модели, при вызове соответствующего метода **Add()** коллекции.

В результате выполнения метода **GetRows()** могут возникать ошибки ввода/вывода, о которых мы узнаем из свойства **LastError** объекта сервера.

Вышеуказанный алгоритм считывания и восстановления данных применяется ко всем сущностям модели до конца выполнения метода **LoadFromBase()**.

Метод **GetRows()** сервера выглядит следующим образом:

/// <summary>

/// Получение набора данных из таблицы

/// </summary>

/// <param name="table">Имя таблицы</param>

/// <param name="likefield">Имя поля для фильтра</param>

/// <param name="text2find">Значение для фильтра</param>

/// <returns></returns>

public DataSet GetRows(string table, string likefield = null, string text2find = null)

{

using (var con = new SqlConnection(Connection))

{

var sql = BuildQuery(table, likefield, text2find);

using (var da = new SqlDataAdapter(sql, con))

{

var ds = new DataSet();

try

{

da.Fill(ds, table);

LastError = "";

}

catch (Exception ex)

{

LastError = ex.Message;

}

return ds;

}

}

}

В методе **GetRows()** создается соединение при помощи вызова конструктора системного объекта **SqlConnection()**, затем создается адаптер при помощи вызова конструктора системного объекта **SqlDataAdapter()**, посредством которого заполняется вновь создаваемый набор данных **DataSet**. Текст sql-скрипта, передаваемого как параметр в конструктор адаптера данных, формируется методом **BuildQuery()** с параметром – именем таблицы.

/// <summary>

/// Построение текста запроса SELECT

/// </summary>

/// <param name="table">Имя таблицы</param>

/// <param name="likefield">Имя поля для фильтра</param>

/// <param name="text2find">Значение для фильтра</param>

/// <returns></returns>

private string BuildQuery(string table, string likefield = null, string text2find = null)

{

var sql = string.Format("SELECT \* FROM [{0}]", table);

if (!string.IsNullOrWhiteSpace(likefield) && !string.IsNullOrWhiteSpace(text2find))

sql += string.Format(" WHERE ([{0}] LIKE N'{1}%')", likefield, text2find);

return sql;

}

Метод **BuildQuery()** запрашивает все поля таблицы. Два необязательных параметра метода могут быть использованы для передачи значения фильтра по текстовому значению указываемого поля.

Для того, чтобы вызов метода **LoadFromBaseAsync()** не «замораживал» интерфейс пользователя, тело метода помещено в вызов задачи (*Task.Run()*) для формирования дочернего потока выполнения кода. После завершения выполнения загрузки данных выдается сообщение в статусную строку. Для того, чтобы синхронизировать работу дочернего потока с работой пользовательского интерфейса, применяется конструкция вида:

var method = new MethodInvoker(() =>

{

// обращение к компонентам формы...

});

if (InvokeRequired)

BeginInvoke(method);

else

method();

Метод ShowDefault(), вызываемый перед асинхронной загрузкой данных модели, показывает начальную «заставку» программы с картинкой гостиницы:

/// <summary>

/// Показ заставки по умолчанию при первой загрузке программы и при смене пользователя

/// </summary>

private void ShowDefault()

{

var rc = new DefaultControl() { Dock = DockStyle.Fill };

CreateAndShowUserControl(rc);

}

В теле этого метода создается необходимый пользовательский элемент управления, в настройках которого указывается, что он должен занимать всю площадь на панели для своего размещения. Далее вызывается общий метод **CreateAndShowUserControl()**, который делает переключение пользовательских компонентов. Вновь размещаемый компонент добавляется в панели отображения **panelView**, а затем предыдущий компонент удаляется, делая процесс переключения форм более плавным и без мерцания.

/// <summary>

/// Размещение контрола поверх старого и удаление старого контрола

/// </summary>

/// <param name="rc"></param>

private void CreateAndShowUserControl(UserControl rc)

{

panelView.Controls.Add(rc);

if (panelView.Controls.Count > 1)

panelView.Controls.RemoveAt(0);

}

После того, как данные модели загружены, для начала работы необходимо «войти» в программу, выбрав фамилию пользователя и ввести пароль. Для этого в главном меню необходимо выбрать пункт «Вход…» для вызова окна «Вход в программу»:

/// <summary>

/// Обработчик пункта меню "Вход..."

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void tsmiLogin\_Click(object sender, EventArgs e)

{

// если есть хотя бы один администратор, записанный в модели данных

if (\_hotel.IsAdministratorDefined())

{

// вызываем форму регистрации, передавая в конструктор ссылку на модель данных

var frm = new LoginForm(\_hotel);

var users = \_hotel.GetUsers(); // список пользователей получаем соответствующим методом

frm.Build(users); // форма настраивает свои компоненты, используя передаваемый в метод этот список

// показываем форму в диалогом режиме

if (frm.ShowDialog(this) == DialogResult.OK)

{

// если пользователь нажал кнопку "ОК"

\_allowedOperations = frm.AllowedOperations; // получаем набор прав пользователя

\_hotel.CurrentUser = frm.User; // получаем ссылку на выбранного пользователя

ShowDefault(); // показываем "заставку"

tsslStatusLabel.Text = ""; // и очищаем статусную строку

}

}

else // нет администраторов в системе

{

\_allowedOperations = AllowedOperations.All; // назначаем полный доступ

\_hotel.CurrentUser = null; // сброс текущего пользователя

// сообщаем пользователю это:

MessageBox.Show("Вам доступны все функции программы, пока не будет определён пользователь с административными правами");

}

}

Сначала проверяем, что есть «настроенные» пользователи с «административными» правами в программе при помощи вызова метода модели **IsAdministratorDefined()**.

/// <summary>

/// Администратор в системе определён

/// </summary>

/// <returns></returns>

public bool IsAdministratorDefined()

{

return GetRegistrators().Count > 0;

}

/// <summary>

/// Получить список сотрудников, имеющих право регистрировать клиентов

/// </summary>

/// <returns></returns>

public List<Employee> GetRegistrators()

{

return RegistryStaff.Where(item => GetEmployeeRole(item.IdEmployeeRole)

.AllowedOperations.HasFlag(AllowedOperations.ManageClients))

.OrderBy(item => string.Concat(item.Surname,item.Name,item.LastName)).ToList();

}

Собственно, метод **IsAdministratorDefined()** вызывает другой метод **GetRegistrators()**, который выдает список сотрудников, имеющих право управлять списком клиентов.

Для этого просматривается коллекция записей сотрудников **RegistryStaff**, для каждой записи по идентификатору должности **IdEmployeeRole** получаем ссылку на объект «должности» методом модели **GetEmployeeRole()** и определяется наличие флага **AllowedOperations.ManageClients** в коллекции прав **AllowedOperations** «должности» при помощи системного метода **HasFlag()**.

Если полученный список не пуст, метод **IsAdministratorDefined()** возвратит **true**.

Далее создается объект формы «Вход в программу», которому в метод **Build()** передается список пользователей, полученный вызовом метода модели **GetUsers()**.

/// <summary>

/// Получить список сотрудников, имеющих право работать с программой

/// </summary>

/// <returns></returns>

public List<Employee> GetUsers()

{

var list = new List<Employee>();

list.AddRange(GetAdministrators());

list.AddRange(GetRegistrators());

return list.OrderBy(item => string.Concat(item.Surname, item.Name, item.LastName)).Distinct().ToList();

}