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 реализацию 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 **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 панель 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();

}

Как можно видеть, список пользователей формируется из списка администраторов и списка регистраторов. Список регистраторов извлекается из списка сотрудников, имеющих право управлять записями клиентов и формированием таблиц «бронирования», «доставки» и тому подобное.

Теперь рассмотрим, как работает метод **Build()** формы «Вход в программу»:

public void Build(List<Employee> list)

{

cbUsers.Items.Clear();

foreach (var user in list)

{

cbUsers.Items.Add(user);

}

}

Здесь всё просто – комбобокс пользователей **cbUsers** очищается и все объекты **Employee** из списка в параметрах метода добавляются в этот комбобокс. Для того, чтобы добавленный в **cbUsers** объект отображался в виде фамилии пользователя и его должности, в классе сотрудника **Employee** перегружен метод преобразования объекта в строку:

public override string ToString()

{

var role = \_hotel.GetEmployeeRole(IdEmployeeRole);

return string.Format($"{Surname} {Name} {LastName} ({role})");

}

Здесь ссылка на объект «должности» получается при вызове метода модели **GetEmployeeRole()** по идентификатору **IdEmployeeRole**, хранящемуся в самом объекте сотрудника, а текстовое представление самой «должности» получается благодаря перегрузке метода преобразования объекта в строку в классе **EmployeeRole**:

public override string ToString() { return NameRole; }

К событию выбора текущего значения в списке выбора **cbUsers** подключен обработчик:

private void cbUsers\_SelectionChangeCommitted(object sender, System.EventArgs e)

{

btnOk.Enabled = !string.IsNullOrWhiteSpace(tbPassword.Text) &&

cbUsers.SelectedItem != null;

UpdateData();

}

Здесь разрешается нажатие кнопки «Ввод», если поле пароля не пустое и выбран какой-нибудь пользователь. В методе **UpdateData()** выбранные в интерфейсе значения передаются публичным свойствам формы:

private void UpdateData()

{

if (cbUsers.SelectedItem != null)

{

var employee = (Employee)cbUsers.SelectedItem;

AllowedOperations = \_hotel.GetEmployeeRole(employee.IdEmployeeRole).AllowedOperations;

User = employee;

PasswordHash = employee.Password;

}

}

Свойство формы **AllowedOperations** содержит права выбранного пользователя, свойство **Users** содержит ссылку на выбранного сотрудника, свойство **PasswordHash** – «хэш» пароля, хранящегося в модели и полученного из базы данных. Сами пароли в базе данных не хранятся в открытом виде, а хранятся их когда-то вычисленные «хэши», а когда приходит время сравнить пароли, то сравниваются их «хэши».

К кнопке «Ввод» привязан обработчик нажатия, в котором проверяется соответствие введённого и запомненного паролей для выбранного пользователя. Если пароль не совпал, то выдается сообщение и закрытия формы не происходит:

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

{

if (Helper.GetHashString(tbPassword.Text) == PasswordHash)

{

DialogResult = DialogResult.OK;

Close();

}

else

MessageBox.Show("Пароль не верный!");

}

Вычисление «хэша» пароля производится в методе статического класса **Helper.GetHashString()** системными функциями:

/// <summary>

/// Класс - помощник

/// </summary>

public static class Helper

{

public static byte[] GetHash(string inputString)

{

HashAlgorithm algorithm = SHA256.Create();

return algorithm.ComputeHash(Encoding.UTF8.GetBytes(inputString));

}

public static string GetHashString(string inputString)

{

StringBuilder sb = new StringBuilder();

foreach (byte b in GetHash(inputString))

sb.Append(b.ToString("X2"));

return sb.ToString();

}

}

Теперь, когда текущий пользователь выбран, пункты главного меню становятся доступными для выбора в зависимости от значения поля **\_allowedOperations** основной формы, а разрешение пунктов делается в обработчиках пунктов меню **tsmiReception** и **tsmiTuning**.

/// <summary>

/// Разрешение пунктов меню Регистратура

/// </summary>

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

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

private void tsmiReception\_DropDownOpening(object sender, EventArgs e)

{

tsmiClients.Enabled = tsmiReservations.Enabled = tsmiTransfer.Enabled =

tsmiAccordancePayChannels.Enabled = tsmiSaveToBase.Enabled =

\_allowedOperations.HasFlag(AllowedOperations.ManageClients);

}

/// <summary>

/// Разрешение пунктов меню Настройка

/// </summary>

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

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

private void tsmiTuning\_DropDownOpening(object sender, EventArgs e)

{

tsmiEmployees.Enabled = tsmiTuningRooms.Enabled = tsmiCategories.Enabled =

tsmiServices.Enabled = tsmiEmployeeRoles.Enabled = tsmiPayChannelsTV.Enabled =

tsmiSaveToFile.Enabled = tsmiLoadFromFile.Enabled =

\_allowedOperations.HasFlag(AllowedOperations.ManageEmployees);

}

Здесь флаг **AllowedOperations.ManageClients** отвечает за управление клиентами, а флаг **AllowedOperations.ManageEmployees** – за управление сотрудниками и словарями данных.

Для работы с программой сначала необходимо определить значения для словарей данных, которые используются для выбора перечисляемых значений в основных рабочих таблицах. Поэтому сначала необходимо заполнить словари данных:

* **EmployeeRoles** – должности сотрудников гостиницы.
* **Categories** – категории комфортности номеров гостиницы.
* **Services** – перечень подключаемых услуг в номерах.
* **PayChannels** – перечень подключаемых платных каналов кабельного телевидения.

Затем необходимо заполнить основные служебные таблицы:

* **RegistryStaff** – список сотрудников гостиницы.
* **Rooms** – список номеров гостиницы.

Теперь, когда служебная информация подготовлена, можно приступать к регистрации постояльцев и бронированию номеров. Заполняем рабочие таблицы:

* **Clients** – список клиентов гостиницы.
* **Reservations** – таблица записей о бронировании номеров.
* **Transfers** – таблица записей заказа транспорта для доставки клиентов из крупных транспортных узлов (вокзалов, аэропортов) к подъезду гостиницы.
* **AccordancePayChannels** – таблица оформления подписки на платные каналы кабельного телевидения.

Рассмотрим работу алгоритмов программы на примере заполнения, изменении и удалении записей в таблице бронирования номеров **Reservations**.

При выборе пункта главного меню «Регистратура\Бронирование номеров» происходит вызов обработчика выбора:

/// <summary>

/// Бронирование номеров

/// </summary>

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

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

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

{

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

rc.Build(\_hotel);

CreateAndShowUserControl(rc);

}

Сначала создаётся новый экземпляр класса пользовательского компонента управления **ReservationsControl**, затем вызывается его метод **Build()** для заполнения данными размещённой на нём компонента-таблицы **DataGridView**:

public void Build(Hotel hotel)

{

\_hotel = hotel;

\_reservations = \_hotel.Reservations;

FillTable();

}

где метод **FillTable()** делает следующее:

private void FillTable()

{

tsbChangeReservationClient.Enabled = tsbDepartureClient.Enabled = false;

// устанавливаем размер виртуальной таблицы

dgvReservations.RowCount = \_reservations.Count;

// просим перерисовать таблицу

dgvReservations.Invalidate();

}

Запрещаются кнопки изменения и удаления текущей выбранной строки. Счётчику строк **DataGridView** присваивается количество элементов в коллекции **Reservations** и вызывается метод требования перерисовки **DataGridView**, который настроен на работу в виртуальном режиме. Это значит, что его свойство **VirtualMode** установлено в **true**, а для получения данных для отображения в «сетке» подключен обработчик **CellValueNeeded()**:

private void dgvReservations\_CellValueNeeded(object sender, DataGridViewCellValueEventArgs e)

{

var sorted = \_reservations;

if (sorted.Count == 0) return;

// для каждого столбца виртуальной таблицы

switch (e.ColumnIndex)

{

case 0: // фамилия, имя и отчество клиента

var clientId = sorted[e.RowIndex].IdClient;

var client = \_hotel.GetClient(clientId);

e.Value = client != null

? string.Format($"{client.Surname} {client.Name} {client.LastName}")

: "нет данных";

break;

case 1: // информация о номере

var roomId = sorted[e.RowIndex].IdRoom;

var room = \_hotel.GetRoom(roomId);

var category = \_hotel.Categories[room.IdCategory];

e.Value = room != null

? string.Format($"{room.NumberSeat}-х местный \"{category.NameCategory}\" №{room.RoomNumber} на {room.Floor}-м этаже")

: "нет данных";

break;

case 2: // дата заезда

e.Value = sorted[e.RowIndex].ArrivalDate.ToShortDateString();

break;

case 3: // дата выезда

e.Value = sorted[e.RowIndex].DepartureDate.ToShortDateString();

break;

case 4: // фамилия, имя и отчество сотрудника

var employeeId = sorted[e.RowIndex].IdEmployee;

var employee = \_hotel.GetEmployee(employeeId);

e.Value = employee != null

? string.Format($"{employee.Surname} {employee.Name} {employee.LastName}")

: "нет данных";

break;

}

}

Для каждого столбца мы определяем значение для **e.Value**, основываясь на индексе столбца **e.ColumnIndex** и индексе строки **e.RowIndex**.

Для добавления нового «бронирования» нажимаем кнопку «Бронирование»:

/// <summary>

/// Нажатие кнопки "Бронирование"

/// </summary>

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

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

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

{

var frm = new ArrivalForm(\_hotel); // создаем форму

var arrival = new Reservation(\_hotel) { IdEmployee = \_hotel.CurrentUser.IdEmployee };

frm.Build(arrival); // создаём "пустое" заселение и заполняем контролы формы

// показываем форму в диалоге

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

{

arrival = frm.Data; // получаем измененные данные заселения

\_reservations.Add(arrival); // добавляем в список бронирования

FillTable(); // перестраиваем таблицу

}

}

Сначала создаём форму «Заселение», затем создаём пустой объект типа **Reservation**, у которого определяем сразу поле **IdEmployee** – идентификатор текущего пользователя, сотрудника гостиницы. Затем передаём этот объект методу **Build()** формы. Показываем форму в модальном режиме для редактирования. После закрытия формы переданный настроенный объект «заселения» через публичное свойство формы **Data** добавляем в список бронирования. Вызываем метод **FillTable()** для показа новых данных.

Посмотрим, как устроен метод **Build()** формы:

//занесение данных из объекта данных в контролы

public void Build(Reservation data)

{

Data = data;

updating++; //включаем режим обновления

BuildTree();

// присваиваем текущее значение списка постояльцев

foreach (var item in cbClientFullName.Items.Cast<Client>())

{

if (item.IdClient == data.IdClient)

{

cbClientFullName.SelectedItem = item;

break;

}

}

// присваиваем текущее значение из списка сотрудников

lbEmployeeFullName.Text = \_hotel.GetEmployee(data.IdEmployee).ToString();

// если комната выбрана

if (cbClientFullName.SelectedItem != null)

{

// день заезда

dtpArrivalDate.Value = data.ArrivalDate.Date;

// день выезда

dtpDepartureDate.Value = data.DepartureDate.Date;

\_room = \_hotel.Rooms.Get(data.IdRoom);

if (\_room != null)

{

UpdateRoomNumber(\_room);

UpdateServicesChecklistbox(\_room.Services);

UpdateArrivalPrice(\_room);

var node = \_index[\_room];

node.EnsureVisible();

tvRooms.SelectedNode = node;

tvRooms.Select();

}

}

else

{

// при создании заселения выставляется текущая дата

dtpArrivalDate.Value = DateTime.Now.Date;

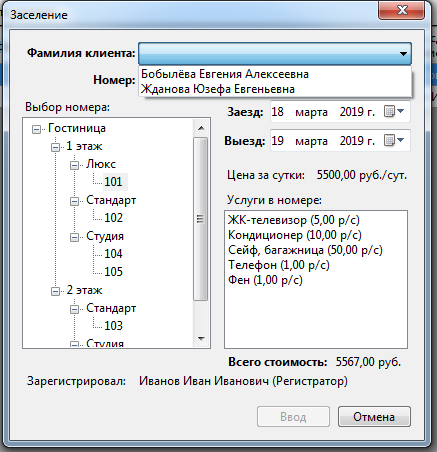
dtpDepartureDate.Value = DateTime.Now.AddDays(1).Date;

}

updating--; //выключаем режим обновления

}

Данные о «заселении» передаются через параметр метода и запоминаются в локальной переменной **data**. Приращение переменной **updating** в начале выполнения метода и уменьшение в конце служит для блокирования работы метода **UpdateData()** в режиме обновления данных в визуальных компонентах, чтобы не вызывать зацикливания работы программы.

Далее заполняем компонент **TreeView** в методе **BuildTree()**.

Посмотрим, как строится «дерево» номеров в методе **BuildTree()**:

private void BuildTree()

{

\_index.Clear();

try

{

tvRooms.AfterSelect -= tvRooms\_AfterSelect;

tvRooms.BeginUpdate();

tvRooms.Nodes.Clear();

// добавим корневой узел

var hotelNode = new FloorTreeNode("Гостиница");

tvRooms.Nodes.Add(hotelNode);

// сформируем группы номеров по этажам

foreach (var floors in \_hotel.Rooms.OrderBy(f => f.Floor).GroupBy(f => f.Floor))

{

// добавляем узлы этажей

var floorNode = new TreeNode(string.Format($"{floors.Key} этаж"));

hotelNode.Nodes.Add(floorNode);

foreach (var categories in floors.GroupBy(c => c.IdCategory).OrderBy(c => \_hotel.Categories[c.Key].NameCategory))

{

// добавляем узлы категорий

var categoryNode = new TreeNode(\_hotel.Categories[categories.Key].NameCategory);

floorNode.Nodes.Add(categoryNode);

foreach (var room in categories.OrderBy(r => r.RoomNumber))

{

// количество занятых мест в номере

var count = \_hotel.RoomUsed(room, dtpArrivalDate.Value, dtpDepartureDate.Value);

var status = room.NumberSeat == count

? "(занят)"

: count != 0

? string.Format($"(ещё: {room.NumberSeat - count})")

: "";

// добавляем узлы номеров

var roomNode = new TreeNode(string.Format($"{room.RoomNumber} {status}")) { Tag = room };

\_index.Add(room, roomNode);

categoryNode.Nodes.Add(roomNode);

}

}

}

}

finally

{

tvRooms.EndUpdate();

tvRooms.AfterSelect += tvRooms\_AfterSelect;

}

// раскрываем все узлы этажей

tvRooms.ExpandAll();

}

Три вложенных цикла **foreach** формируют вложенную структуру узлов дерева, сначала по номерам этажей, потом по категориям номеров и, в последнюю очередь, по номерам комнат. Группировка по номерам этажей делается для коллекции номеров **Rooms**, последующие группировки делаются для получившихся групп.

Далее выполняется поиск в списке клиентов **cbClientFullName**, по совпадению свойств **IdClient** элемента списка и параметра **data**.

Так как метод **Build()** формы может вызываться и для новой записи и для существующей записи, то определение, с какой записью работаем, определяем по условию **cbClientFullName.SelectedItem != null**. Если ссылка не пустая, то выбранный элемент списка существует. В этом случае заполняем дату приезда и дату отъезда, получаем ссылку на комнату методом модели **Rooms.Get()**, потом заполняем список подключённых к номеру услуг (*список определяется при конфигурировании номера*) и рассчитывается окончательная стоимость номера за период заселения в методе **UpdateArrivalPrice()**.

Текущий выбранный узел в дереве комнат выбирается через индекс узлов **\_index[]**, созданный при построении дерева в методе **BuildTree().**

Если запись о заселении новая, то датой приезда назначается текущий день, а датой отъезда – следующий день.

При выборе строки в таблице «бронирования» становятся доступными кнопки «Изменить данные» и «Удалить данные».

При нажатии на кнопку «Изменить данные» выполняется подключенный обработчик нажатия кнопки **tsbChangeReservationClient\_Click**:

/// <summary>

/// Нажатие кнопки "Изменить данные"

/// </summary>

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

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

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

{

var frm = new ArrivalForm(\_hotel); // создаем форму

frm.Build(\_reservations[dgvReservations.SelectedRows[0].Index]); // заполняем контролы формы параметрами выбранного заезда

// вызываем форму на редактирование

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

{

// обновляем дерево этажей и виртуальную таблицу

FillTable(); // перестраиваем таблицу

}

}

В метод **Build()** формы передаётся запись о бронировании, полученная через текущий индекс сетки **dgvReservations** из коллекции **Reservations** модели (*сетка настроена на работу в виртуальном режиме*). После закрытия формы диалога обновляем содержимое сетки.

При нажатии на кнопку «Удалить данные» выполняется следующий обработчик:

/// <summary>

/// Нажатие кнопки "Удалить данные"

/// </summary>

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

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

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

{

if (MessageBox.Show(this, "Удалить информацию о заселении?", "Удаление брони",

MessageBoxButtons.YesNo, MessageBoxIcon.Question) == DialogResult.Yes)

{

var reservation = \_reservations[dgvReservations.SelectedRows[0].Index];

try

{

\_hotel.CheckReservationUsed(reservation);

// удаляем заселение из списка заседений

\_reservations.Remove(reservation);

// обновляем данные интерфейса

FillTable();

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

}

После запроса на подтверждение удаления информации получаем ссылку на объект бронирования из коллекции **Reservations** модели через локальное поле **\_reservations[]**. Для того, чтобы не удалить нужные данные, вызываем метод модели **CheckReservationUsed()**, который проверяет наличие существующих ссылок в других коллекциях на эту запись и вызывает программное исключение в этом случае. Если исключения не произошло, то выполняется метод **Remove()** для коллекции бронирования, а затем сетка обновляется.

Если исключение произошло, то оно фиксируется в блоке **try/catch** и выдаётся сообщение об этом.

Посмотрим, как устроен метод **CheckReservationUsed()**:

/// <summary>

/// Проверить на использование бронирования

/// </summary>

/// <param name="reservation"></param>

public void CheckReservationUsed(Reservation reservation)

{

if (Transfers.Any(item => item.IdReservation == reservation.IdReservation) ||

AccordancePayChannels.Any(item => item.IdReservation == reservation.IdReservation))

throw new Exception("Эти данные о бронировании ещё используются!");

}

Записи о бронировании могут использоваться в коллекции трансферов или в коллекции подписок на платные каналы, поэтому проверяем существование ссылки в обоих коллекциях и, если находим, то создаём исключение с соответствующим сообщением.

Работа с остальными таблицами или справочниками устроена аналогичным образом, поэтому перейдём к рассмотрению алгоритмов формирования отчётов.

Отчёты можно вызвать, выбрав пункты главного меню «Отчёты». При выборе пункта меню «Отчёты\За последний месяц\Список постояльцев» выполняется следующий обработчик:

/// <summary>

/// Вызов отчёта по клиентам за прошедший месяц

/// </summary>

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

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

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

{

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

var generator = new ReportsBuilder();

var report = generator.GetClientsByLastMonth(\_hotel);

rc.Build(\_hotel, report);

CreateAndShowUserControl(rc);

}

Создаём пользовательский компонент **ReportViewControl**, который будет отображать отчёт в стандартном компоненте **ListView**. Потом создаём экземпляр генератора отчётов при помощи конструктора типа **ReportsBuilder()** и вызываем метод создания отчёта для выдачи списка клиентов, проживавших в гостинице в течение последнего месяца. Созданный отчёт передаём в метод **Build()** пользовательского компонента управления и выводим его на форму ранее описанным методом **CreateAndShowUserControl()**.

Рассмотрим устройство класса генератора отчётов ReportsBuilder:

public class ReportsBuilder

{

/// <summary>

/// Формирование отчёта по свободным номерам на указанную дату

/// </summary>

public Report GetEmptyRoomsByDate(Hotel hotel, DateTime date) {…}

/// <summary>

/// Формирование отчёта по занятым номерам на указанную дату

/// </summary>

public Report GetBusyRoomsByDate(Hotel hotel, DateTime date) {…}

/// <summary>

/// Формирование отчёта по клиентам на указанную дату

/// </summary>

public Report GetClientsByDate(Hotel hotel, DateTime date) {…}

/// <summary>

/// Список постоятельцев за последний месяц

/// </summary>

public Report GetClientsByLastMonth(Hotel hotel) {…}

}

Класс **ReportsBuilder** содержит четыре метода для создания отчётов, результатом работы которых является создание нового экземпляра класса отчёта **Report**:

/// <summary>

/// Отчёт

/// </summary>

public class Report

{

public string Caption { get; set; } // заголовок отчета

public ReportColumns ReportColumns { get; set; } // колонки отчета

public ReportRows ReportRows { get; set; } // строки отчета

}

Объект отчёта содержит его заголовок, коллекцию колонок и коллекцию строк. Коллекция колонок определяется следующим образом:

/// <summary>

/// Колонки (заголовки столбцов) отчета

/// </summary>

public class ReportColumns : List<string>

{

/// <summary>

/// Добавить заголовки

/// </summary>

/// <param name="args"></param>

public void Add(params string[] args)

{

foreach (var item in args)

base.Add(item);

}

}

Коллекция колонок наследуется от списка строк и содержит метод для добавления списка наименований колонок удобным способом.

Коллекция строк определяется следующим образом:

/// <summary>

/// Строки отчета

/// </summary>

public class ReportRows :List<ReportRow>

{

// добавить строку отчета

public void Add(params string[] args)

{

var row = new ReportRow();

row.Add(args);

base.Add(row);

}

}

/// <summary>

/// Строка отчета

/// </summary>

public class ReportRow

{

public List<string> Items { get; set; } = new List<string>(); // значения в колонках

/// <summary>

/// Добавить значения для колонок строки

/// </summary>

/// <param name="args"></param>

public void Add(params string[] args)

{

foreach (var item in args)

Items.Add(item);

}

}

Коллекция строк является списком объектов строки, которые сами по себе являются списками строк.

Теперь посмотрим, как метод **GetClientsByLastMonth()** строит отчёт:

/// <summary>

/// Список постояльцев за последний месяц

/// </summary>

/// <param name="hotel"></param>

/// <returns></returns>

public Report GetClientsByLastMonth(Hotel hotel)

{

var report = new Report

{

Caption = "Список постояльцев за последний месяц"

};

report.ReportColumns.Add("Фамилия клиента", "Номер комнаты", "Въезд", "Выезд");

var last = DateTime.Now;

var first = new DateTime(last.Year, last.Month, 1);

// заполнение строк отчёта

foreach (var item in hotel.Reservations

.Where(item => item.ArrivalDate >= first && item.ArrivalDate <= last ||

item.DepartureDate >= first && item.DepartureDate <= last)

.OrderBy(item => hotel.GetClient(item.IdClient).Surname))

{

var client = hotel.GetClient(item.IdClient);

var room = hotel.GetRoom(item.IdRoom);

var category = hotel.Categories[room.IdCategory];

report.ReportRows.Add($"{client.Surname} {client.Name} {client.LastName}", $"{room.RoomNumber} ({category})",

item.ArrivalDate.ToShortDateString(), item.DepartureDate.ToShortDateString());

}

return report;

}

Сначала создаём «пустой» объект отчёта **Report** и заполняем значение его свойства **Caption** – это будет заголовок отчёта. Затем добавляем четыре наименования колонок отчёта.

Диапазон дат, для которого будем делать выборку, задаём «задом наперёд». Конечная дата – это текущая дата, а начальная дата строится по году и месяцу от текущей даты и на первый день месяца.

В цикле **foreach** из коллекции бронирования **Reservations** делаем выборку, чтобы начальная и конечная дата заселения были в диапазоне поиска. Для найденных записей получаем ссылки на клиента, номер и категорию номера. Используем ссылки на объекты модели для добавления новой строки отчёта.

Созданный отчёт передаётся методу **Build()** пользовательского компонента управления **ReportViewControl** для показа. Вот, как это происходит:

public void Build(Hotel hotel, Report report)

{

\_hotel = hotel;

\_report = report;

lbReportCaption.Text = report.Caption;

lvReport.Columns.Clear();

lvReport.Items.Clear();

foreach (var column in report.ReportColumns)

{

lvReport.Columns.Add(column, 200);

}

foreach (var row in report.ReportRows)

{

var lvi = new ListViewItem

{

Text = row.Items[0]

};

foreach (var item in row.Items.Skip(1))

lvi.SubItems.Add(item);

lvReport.Items.Add(lvi);

}

}

Столбцы и строки компонента **ListView** очищаются, чтобы заполнить их динамически. Столбцы **ListView** заполняются из коллекции столбцов отчёта. Для столбцов указывается фиксируемая ширина в 200 единиц, что в общем случае вполне достаточно. Далее заполняются строки компонента **ListView**.

В заключение рассмотрим алгоритм записи данных модели в базу данных. Для этой задачи создан метод **SaveToBase()** статического класса **SaverLoader** модели. Этот метод вызывается в обработчике нажатия пункта меню «Сохранить данные» или при закрытии основной формы, в методе **SaveToBase()** основной формы:

/// <summary>

/// Сохранение данных модели в базу данных

/// </summary>

private void SaveToBase()

{

SaverLoader.SaveToBase(Properties.Settings.Default.ConnectionString, \_hotel);

var method = new MethodInvoker(() =>

{

var result = SaverLoader.OperationResult;

tsslStatusLabel.Text = string.IsNullOrWhiteSpace(result)

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

statusStrip1.Refresh();

});

if (InvokeRequired)

BeginInvoke(method);

else

method();

}

Метод формы **SaveToBase()** может вызываться асинхронно, поэтому для выдачи сообщения после окончания его работы используется проверка условия **InvokeRequired**. Метод модели **SaveLoader.SaveToBase()** устроен следующим образом:

/// <summary>

/// Метод сохранения конфигурации в базу данных

/// </summary>

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

/// <param name="hotel"></param>

public static void SaveToBase(string connection, Hotel hotel)

{

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

// категории

server.DeleteInto("Categories", "IdCategory", hotel.Categories.Select(item => item.Value.IdCategory));

OperationResult = server.LastError;

foreach (var item in hotel.Categories.Values)

{

var columns = new Dictionary<string, string>

{

{ "IdCategory", item.IdCategory.ToString() },

{ "NameCategory", item.NameCategory }

};

if (!server.InsertInto("Categories", columns)) server.UpdateInto("Categories", columns);

OperationResult = server.LastError;

}

// услуги

server.DeleteInto("Services", "IdService", hotel.Services.Select(item => item.IdService));

// …

}

Вначале создаём объект типа **SqlServer**, который содержит методы **InsertInto()**, **UpdateInto()** и **DeleteInto()** для работы с SQL-сервером.

Метод **DeleteInto()** получает в качестве параметров имя таблицы на сервере, имя ключевого поля и список ключей соответствующей коллекции модели данных, которые актуальны на данный момент. Метод формирует SQL-скрипт, который удаляет из базы данных те записи, ключей которых нет в передаваемом списке ключей. Это позволяет удалять в базе данных неактуальные версии записей коллекции.

Устройство метода **DeleteInto()**:

/// <summary>

/// Удаление записей таблицы, которых больше нет в модели, а есть в базе данных

/// </summary>

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

/// <param name="keyfield">Ключевой столбец</param>

/// <param name="modelKeyList">Спискок ключей в модели</param>

/// <returns></returns>

public bool DeleteInto(string table, string keyfield, IEnumerable<Guid> modelKeyList)

{

var ds = GetRows(table);

if (ds.Tables.Count == 0) return true;

var fordelete = new List<Guid>(); // список ключевых значений для удаления

// получаем ключи из базы данных

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

{

var key = Guid.Parse(row.ItemArray[0].ToString());

// если ключ базы данных отсутствует в списке ключей модели

if (!modelKeyList.Contains(key))

fordelete.Add(key); // то добавляем его в список

}

if (fordelete.Count == 0) return true;

using (var con = new SqlConnection(Connection))

{

try

{

con.Open();

// удаление выбранных ключей

foreach (var key in fordelete)

{

var sql = string.Format("DELETE FROM [{0}] WHERE [{1}]=N'{2}'", table, keyfield, key);

using (SqlCommand cmd = new SqlCommand(sql, con))

{

cmd.ExecuteNonQuery();

}

}

con.Close();

LastError = "";

return true;

}

catch (Exception ex)

{

LastError = ex.Message;

return false;

}

}

}

Далее для каждой записи в коллекции вызывается метод **InsertInto()**, а если он возвращает false, то метод **UpdateInto()**.

Устройство метода **InsertInto()**:

public bool InsertInto(string table, Dictionary<string, string> columns)

{

using (var con = new SqlConnection(Connection))

{

try

{

con.Open();

// формирование запроса для вставки

var names = new List<string>();

var values = new List<string>();

foreach (var key in columns.Keys)

{

names.Add("[" + key + "]");

values.Add("N'" + columns[key] + "'");

}

var sql = string.Format("INSERT INTO [{0}] ({1}) VALUES({2})",

table.ToLower(), string.Join(", ", names), string.Join(", ", values));

using (SqlCommand cmd = new SqlCommand(sql, con))

{

cmd.ExecuteNonQuery();

}

con.Close();

LastError = "";

return true;

}

catch (Exception ex)

{

LastError = ex.Message;

return false;

}

}

}

Устройство метода **UpdateInto()**:

/// <summary>

/// Запрос на изменение данных

/// </summary>

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

/// <param name="columns">Набор данных для изменения</param>

/// <returns></returns>

public bool UpdateInto(string table, Dictionary<string, string> columns)

{

using (var con = new SqlConnection(Connection))

{

try

{

con.Open();

// формирование запроса для изменения

var values = new List<string>();

var indexName = columns.Keys.First();

var indexValue = columns[indexName];

foreach (var key in columns.Keys.Skip(1))

{

values.Add("[" + key + "] = N'" + columns[key] + "'");

}

var sql = string.Format("UPDATE [{0}] SET {1} WHERE [{2}]=N'{3}'",

table.ToLower(), string.Join(", ", values), indexName, indexValue);

using (SqlCommand cmd = new SqlCommand(sql, con))

{

cmd.ExecuteNonQuery();

}

con.Close();

LastError = "";

return true;

}

catch (Exception ex)

{

LastError = ex.Message;

return false;

}

}

}

Применение последовательно вызовов **InsertInto()** и **UpdateInto()** связано с тем, что если запись в базе уже существует, то выполнение SQL-скрипта «INSERT» приведёт к ошибке выполнения и метод возвратит false. Тогда выполнение SQL-скрипта «UPDATE» внесёт необходимые изменения в базу данных.