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();

}

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 пароль 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 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 (*как пример*).

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

}