**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«РОССИЙСКАЯ АКАДЕМИЯ НАРОДНОГО ХОЗЯЙСТВА**

**И ГОСУДАРСТВЕННОЙ СЛУЖБЫ**

**ПРИ ПРЕЗИДЕНТЕ РОССИЙСКОЙ ФЕДЕРАЦИИ»**

**Смоленский филиал РАНХиГС**

Специальность:

09.02.07 Информационные системы и программирование

**ПРОЕКТНАЯ РАБОТА**

по дисциплине **«**Управление и автоматизация баз данных»

на тему: «Проектирование элементов системы автоматизации продаж музыкального оборудования»

**Автор:**

обучающийся

группы 37/11К-ИТО

очной формы обучения

Махницкий Д.С.

**Руководитель:**

преподаватель

Харламова О.Е.

Смоленск 2024

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc181971278)

[1 Информационное, алгоритмическое и программное обеспечение задачи автоматизации 4](#_Toc181971279)

[2 Тестирование элементов системы и формирование технологической среды 5](#_Toc181971280)

[3 Анализ проблем автоматизации процесса продаж музыкального оборудования и пути их решения 6](#_Toc181971281)

[ЗАКЛЮЧЕНИЕ 7](#_Toc181971282)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 8](#_Toc181971283)

[ПРИЛОЖЕНИЕ А 9](#_Toc181971284)

[ПРИЛОЖЕНИЕ Б 10](#_Toc181971285)

[ПРИЛОЖЕНИЕ В 11](#_Toc181971286)

# ВВЕДЕНИЕ

# 1 Информационное, алгоритмическое и программное обеспечение задачи автоматизации

Согласно предметной области (продажа товара в магазине музыкального оборудования), была смоделирована база данных (БД) с помощью библиотеки Microsoft.EntityFrameworkCore на языке С# и системы управления базами данных (СУБД) MySQL. Данная БД хранит информацию о товаре, который либо находится на складе, либо ожидает поставки, либо продан, либо зарезервирован, либо находится в электронной корзине (логическое место, куда продавец добавляет товары для совершения единой их продажи), либо ожидает оплаты (следующий этап после нахождения в корзине, но до получения статуса продано). На рисунках 1.1, 1.2 и 1.3 изображены концептуальная, логическая и физическая модели базы данных соответственно.

Изображение выглядит как текст, снимок экрана, диаграмма, число

Автоматически созданное описание

Рисунок 1.1 – Концептуальная модель БД в нотации IDEF1X

Изображение выглядит как текст, снимок экрана, диаграмма, число

Автоматически созданное описание

Рисунок 1.2 – Логическая модель БД в нотации Мартина

В логической модели (рисунок 1.2) можно заметить определенную симметрию – каждый вид музыкального товара, а именно аксессуары (accessories), аудио-оборудование (audio\_equipment\_units), нотные издания (sheet\_music\_editions) и музыкальные инструменты (musical\_instruments) имеет собственную таблицу для типов с шаблонным названием имя\_вида\_товара\_specific\_types, а так же свою связующую таблицу для отношения (тоже с шаблонным названием имя\_вида\_товара\_sale) многие-ко-многим между собой и таблицей «продажи» (sales).

Опишем преимущество использования отдельной таблицы со специфичными типами (подтипами) для каждого вида товара. Каждый вид товара имеет уникальные подтипы, которые, в данном случае, могут пересекаться с названиями других подтипов (например, данная структура позволят добавить в БД гитару (музыкальный инструмент) с подтипом «акустика», а также колонки (аудио-оборудование) с таким же подтипом, поддерживая ограничение уникальности названия для каждого вида товара. Из альтернатив данной реализации можно рассмотреть создание единой таблицы со всеми подтипами. В таком случае необходимо было бы создать вторую таблицу со всеми видами товара, и в каждой строке подтипа хранить информацию о виде товара (дискриминатор). Например, строка, которая в текущей реализации выглядит как отображено в таблице 1.1, в альтернативном варианте выглядела бы как в таблице 1.2.

Таблица 1.1 – Пример строки из таблицы musical\_instrument\_specific\_type

|  |  |
| --- | --- |
| specific\_type\_id | name |
| 1 | Акустика |

Таблица 1.2 – Пример строки из альтернативного варианта таблицы подтипов specific\_types

|  |  |  |
| --- | --- | --- |
| specific\_type\_id | name | kind\_of\_goods |
| 1 | Акустика | Музыкальные инструменты |

Таким образом, каждая строка должна была бы хранить лишнюю (с точки зрения текущей реализации) информацию (пренебрегая нормализацией). В текущей реализации вид товара «автоматически» хранится в таблице в соответствии с ее названием. Второй плюс данной реализации – более быстрый запрос подтипов товара определенного вида – достаточно выполнения следующей команды:

SELECT \*

FROM `musical\_instrument\_specific\_types`;

вместо:

SELECT \* `specific\_types`

WHERE `kind\_of\_goods` = ‘Музыкальные инструменты’;

Другими словами, в текущей реализации подтипы уже отфильтрованы, что увеличивает скорость выполнения запроса на получение всех подтипов определенного вида товара.

Для хранения информации о товаре разных типов используется подход Table Per Class (TPC) – для каждого класса (сущности) создается отдельная таблица со всеми свойствами (атрибутами). Объясняется это тем, что в данном случае ожидается, что поиск товара (относительно сложный по своей структуре запрос) будет осуществляться довольно часто, причем всегда по одному виду товара (с точки зрения предметной области нет смысла одновременно пытаться искать музыкальный инструмент и нотное издание (например, по фразе «синтезатор») – при поиске конечный пользователь всегда знает какой вид товара его интересует) и в результате, при поиске товара запрос так же, как в предыдущем случае с подтипами, не будет содержать дополнительной фильтрации по виду товара.

Оба сделанных выбора вариантов хранения информации в БД основаны на эмпирическом методе (так как данная работа – учебная), в реальной информационной системе (ИС) необходимо выполнить тесты для определения лучшего подхода.

Важная особенность данной БД заключается в том, что таблица sales (продажи) имеет отношение с таблицами товаров разного вида «многие-ко-многим», а таблица goods\_delivery (доставка товара) – «один-ко-многим». Рассмотрим два похожих по смыслу случая:

- Клиент магазина купил музыкальный инструмент, но позже вернул его. Затем этот инструмент купил другой клиент.

- Поставщик доставил нотное издание, которое через некоторое время купил определенный клиент, а затем это нотное издание «чудом» (крайне маловероятный случай) оказалось на складе у поставщика, который затем снова выполнит поставку товара с этим нотным изданием.

В результате определенный товар может побывать сразу в двух и более разных покупках и на практике это происходит регулярно. Возврат товара клиентом – обыденное дело. Также книга действительно может побывать в двух и более поставках товара, но вероятность данного случая настолько мала, что ей можно пренебречь. Соответственно, в первом случае реализовано отношение «многие-ко-многим», а во втором – «один-ко-многим».

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 1.3 – Физическая модель базы данных в нотации Мартина

Относительно физической модели БД можно сказать следующее:

- В качестве идентификатором используются глобально уникальные идентификаторы (GUID), длина которых – 36 символов (включая тире) поэтому id всех таблиц имеет тип CHAR(36).

- Атрибут soft\_deleted (досл. мягко удалено) отображает состояние логической удаленности строки (1 – удалено, 0 – не удалено). Данный атрибут отсутствует у связующих таблиц, так как в его наличии у этих таблиц нет смысла.

- Таблицы с информацией о товарах атрибуты с одинаковыми названиями и типами, что в дальнейшем будет отображено на диаграмме классов. При этом некоторые таблицы имеют дополнительный атрибуты. Например, таблицы musical\_instruments и sheet\_music\_edition имеют атрибут release\_year (год выпуска).

- Таблица sales имеет три атрибута типа DATETIME. Согласно предметной области, у продажи должна быть как минимум один из этих атрибутов. Например, зарезервированная продажа будет иметь значение только у атрибута reservation\_date (дата резервирования) и находиться в состоянии ожидания получения клиентом. Наличие значения только у одного из других двух атрибутов так же имеет смысл с точки зрения предметной области. В некоторых случаях продажа может иметь значение сразу у всех трех атрибутов (клиент зарезервировал товар, оплатил (и получил), а затем вернул). Целостность состояния данных трех атрибутов поддерживается с помощью триггеров на внесение и обновление строк.

Назначение ИС – автоматизация продаж музыкального оборудования. Конечные пользователи – персонал магазина, то есть ограниченный круг лиц (от четырех человек). Рассмотрим алгоритмы работы ИС. У каждой категории пользователей есть типичный для нее способ взаимодействия с системой, которым они пользуются большую часть времени:

- продавец ищет и резервирует товар, просматривает о нем информацию, оформляет продажи и возвраты,

- администратор добавляет, удаляет и редактирует существующих пользователей, просматривает отчеты и создает резервные копии БД,

- складской менеджер вносит информацию о новом товаре в ИС,

- консультант ищет товары и просматривает о них информацию.

В данном случае у консультанта меньше всего прав в данной ИС, а все остальные пользователи наделены правами консультанта, соответственно сначала рассмотрим его алгоритм работы (рисунок 1.4).

Изображение выглядит как текст, диаграмма, линия, чек

Автоматически созданное описание

Рисунок 1.4 – типичный способ использования ИС консультантом

Здесь пользователь вводит информацию, которая поможет найти желаемый товар, затем система обрабатывает данный запрос и выводит результаты. Если желаемый товар не был найден, рекомендуется повторить ввод данных для поиска, так как там могла быть ошибка пользователя (например, он выбрал вид «аудио-оборудование», но пытался найти товар по запросу «глюкофон», что является музыкальным инструментом).

Рассмотрим способ использования ИС продавцом (рисунок 1.5), алгоритм работы которого изображен в виде жизненного цикла определенной продажи определенному покупателю: сначала клиент может зарезервировать продажу, затем оплатить ее и получить товар, а затем может вернуть. Как указано в комментариях к блок-схеме, этап резервации и возврата может отсутствовать – например, клиент пришел в магазин, купил товар и больше не возвращался.

Изображение выглядит как диаграмма, текст, Технический чертеж, План

Автоматически созданное описание

Рисунок 1.5 - типичный способ использования ИС продавцом

Алгоритм работы складского менеджера (рисунок 1.6) достаточно тривиален – пользователь вводит данные о товаре, система выполняет валидацию и если она успешна то добавляет товар в ИС, иначе – отображает ошибки.

Изображение выглядит как текст, диаграмма, линия, шаблон

Автоматически созданное описание

Рисунок 1.6 - типичный способ использования ИС складским менеджером

Алгоритм использования ИС администратором не такой структурированный и последовательный, как предыдущие алгоритмы, поэтому его описание будет текстовым. Администратор управляет всеми пользователями в ИС, включая следующие действия:

- просмотр и редактирование полной информации о пользователе,

- добавление или удаление ролей пользователя,

- добавление или удаление пользователей.

Относительно отчетности администратору доступны следующие действия:

- генерация отчетов в соответствии с желаемыми параметрами,

- просмотр отчетов.

Действия администратора относительно резервного копирования БД:

- создание логической резервной копии БД с текстовым примечанием,

- восстановление базы данных из резервной копии.

Таким образом, совокупность обобщенно описанных алгоритмов представляет собой описание общего алгоритма работы ИС.

В качество сложного алгоритма ИС можно рассмотреть алгоритм, который оформляет продажу как проданную. Вводная информация об ИС: архитектура – n-уровневая (слоистая), используемый в приложении паттерн проектирования для обработки HTTP-запросов – Model View Controller (MVC). В качестве подготовки для выполнения работы данного алгоритма сначала необходимо добавить минимум одну единицу товара в корзину, как это изображено на рисунке 1.7.

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание

Рисунок 1.7 – Корзина с четырьмя добавленными единицами товара

После нажатия кнопки «Оформить продажу» пользователь переходит на страницу оплаты (рисунок 1.8), но перед переходом на эту страницу происходит обработка товара в корзине и создание новой продажи. Кнопку «Оформить продажу» обрабатывает метод CreateSaleAsNotSold контроллера SalesController (листинг 1.1).

Изображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описание

Рисунок 1.8 -

При нажатии на кнопку «Успешно оплачено», пользователь получает сообщение «Успешно оплачено» (рисунок 1.9), товар удаляется из корзины (рисунок 1.10), а в списке продаж появляется только что совершенная продажа (рисунок 1.11).

Изображение выглядит как текст, снимок экрана, Мультимедийное программное обеспечение, мультимедиа

Автоматически созданное описание

Рисунок 1.9 – Сообщение об успешной оплате

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 1.10 – Очищенная корзина после совершения продажи

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

Рисунок 1.11 – Совершенная продажа

Листинг 1.1

[HttpPost("/sale/arrange")]

public async Task<IActionResult> CreateSaleAsNotSold([FromForm] SalePaidBy? paidBy, [FromServices] ICreateSaleService createSaleService, [FromServices] ICartService cartService)

{

var goods = await cartService.GetGoodsFromCart(GoodsIdsAndKinds);

Guid? saleId = await createSaleService.CreateSaleAsNotPaid(goods, paidBy);

if (!createSaleService.HasErrors)

{

ClearCart();

return RedirectToAction("PayForSale", new { saleId });

}

else

return RedirectToAction("Cart", "Goods"); //следовало вывести ошибки, Но Это Потом (НЭП)

}

Данный метод получает товар из корзины в виде объектов, создает продажу с этими товарами (со статусом ожидает оплаты). При отсутствии ошибок (что проверяется с помощью свойства ICreateSaleService.HasErrors и объясняется далее) происходит очистка корзины, так как теперь товар привязан к определенной продаже, а затем происходит перенаправление на страницу оплаты (через метод GET, в качестве query-string параметра передается saleId с помощью анонимного объекта с использованием инициализатора проекций – свойство объекта получает имя текущей переменной (или свойства), переданной ему). Для выполнения описанных выше действий в используемом в данном листинге методе ICreateSaleService.CreateSaleAsNotPaid() применяется паттерн, описанный в книге Джона П Смита (John P Smith) «Asp.Net core in action» под названием BuzRunner [JOHN SMITH]. Данный паттерн упрощает обработку сложной бизнес-логики, но усложняет написание простой бизнес-логики, поэтому при разработке ИС он использовался не везде. Для упрощения разработки данной ИС паттерн был перенесен в отдельную библиотеку BizRunner и модифицирован. Итоговая диаграмма классов данного паттерна изображена на рисунке 1.8. Идея заключается в следующем: класс Runner (имеется в виду любой объект данного класса. Можно было создать абстрактный класс Runner, но в этом не было необходимости) вызывает метод Action() у переданного ему реализующий интерфейс IBizAction<in TIn, outTOut> объект, затем проверяет данный объект на наличие ошибок. Если ошибок нет, то Runner сохраняет все изменения, сделанные этим объектом. Метод Runner.Run() выглядит следующим образом:

public TOut Run(TIn argument)

{

var result = action.Action(argument);

if (!action.HasErrors)

context.SaveChanges();

return result;

}

Класс Runner создается один раз и затем используется несколько, в разных сервисах, получая разные объекты действий. Тем не менее, данный класс может иметь разные виды, например, RunnerTransact2WriteDb, который использует два объекта действий вместо одного и выполняет их в виде одной транзакции. В самом методе IBizAction.Action(argument) и упакована бизнес-логика.

Изображение выглядит как текст, снимок экрана, диаграмма, Шрифт

Автоматически созданное описание

Рисунок 1.8

В качестве примера класса, реализующего интерфейс IBizAction<in TIn, out TOut>, можно привести класс UpdateUserAction, а точнее, его метод Action(TIn argument) (листинг 1.2).

Листинг 1.2

public async Task<string?> Action(UpdateUserDto dto)

{ // некоторые свойства можно было валидировать через js, НЭП

var user = await dbAccess.GetUserInfo(dto.Id);

if (user == null)

{

AddError("Пользователь с таким идентификатором не найден", nameof(user.Id));

return null;

}

if (string.IsNullOrWhiteSpace(user.UserName))

AddError("Имя пользователя не может быть пустым", nameof(user.UserName));

else

{

var normalizedUserName = dto.UserName!.ToUpper();

if (!await dbAccess.IsUniqueNormalizedUserName(normalizedUserName, user.Id))

AddError("Пользователь с таким именем уже существует", nameof(user.UserName));

else

{

user.UserName = dto.UserName;

user.NormalizedUserName = normalizedUserName;

}

}

if (dto.EmailConfirmed && string.IsNullOrWhiteSpace(dto.Email))

AddError("Электронная почта указана как подтврежденная, но ее значение отсутствует", nameof(user.EmailConfirmed), nameof(user.Email));

if (!string.IsNullOrWhiteSpace(dto.Email))

{

var normalizedEmail = dto.Email.ToUpper();

if (!await dbAccess.IsUniqueNormalizedEmail(normalizedEmail, user.Id))

AddError("Данная электронная почта уже используется", nameof(user.Email));

else

{

user.Email = dto.Email;

user.NormalizedEmail = normalizedEmail;

user.EmailConfirmed = dto.EmailConfirmed;

}

}

else

{

user.Email = null;

user.NormalizedEmail = null;

user.EmailConfirmed = false;

}

if (dto.PhoneNumber == null)

{

if (dto.PhoneNumberConfirmed)

AddError("Номер телефона указан как подтврежденный, но его значение отсутствует", nameof(user.PhoneNumberConfirmed), nameof(user.PhoneNumber));

}

else if (dto.PhoneNumber.Length != 11)

AddError("Номер телефона указан не полностью", nameof(user.PhoneNumber));

user.PhoneNumber = dto.PhoneNumber;

user.PhoneNumberConfirmed = dto.PhoneNumberConfirmed;

user.TwoFactorEnabled = dto.TwoFactorEnabled;

user.LockoutEnd = dto.LockoutEnd != null ? ((DateTime)dto.LockoutEnd).ToUniversalTime() : null;

user.LockoutEnabled = dto.LockoutEnabled;

return user.Id;

}

Продолжим рассмотрение алгоритма, который оформляет продажу. В листинге 1.3 отображены действия метода, вызываемого сервисом ICreateSaleService (который является Runner’ом). Преимущества описанного ранее паттерна BizRunner в том, что при обнаружении ошибки метод не возвращает значение сразу (кроме случаев, когда в этом есть смысл. Например, здесь если список товаров пуст, то имеет смысл сразу добавить ошибку и вернуть null, так как последующие проверки не добавят ошибок), а выполняет весь остальной код, собирая все остальные ошибки. Так же в данном паттерне плюсом будет упаковать логику взаимодействия с базой данных в отдельный слой чтобы сделать бизнес-логику более чистой и очевидной. Здесь доступ к базе данных осуществляется через readonly свойство класса SalesDbAccess dbAccess. Важный момент при использовании данного класса заключается в том, чтобы не вызывать метод context.SaveChanges(), так как он должен быть вызван внешним кодом (Runner’ом). В идеале, согласно SoC принципу, необходимо разделить доступ к базе данных для сложной и простой бизнес логики, вызывая метод SaveChanges() для простой логики и не вызывая для сложной (которая реализована с помощью BizRunner), что можно сделать, например, с помощью паттерна «Адаптер», разрешая для простой бизнес-логике выбирать между сохранить или нет, а для сложной – запретить данный выбор.

Продолжая говорить о листинге 1.3, метод Action в нем выполняет «сложную» валидацию, которая проверяет значения свойств (а не их корректность с точки зрения типов данных – валидация модели уже выполнена к этому моменту) и их целостность относительно предметной области. Так, например, если происходит оформление продажи с определенным списком товаров, то статус этих товаров должен быть «в корзине», они не должны быть логически удаленными из БД или товар не может быть оформлен на продажу если он уже относится к определенной продаже (не зарезервированной). При успешной валидации создается объект продажи, которому присваивается текущие дата и время, а объект dbAccess связывает эту продажу со списком товаров.

Листинг 1.3

public async Task<Guid?> Action(CreateSaleDto dto)

{

if (dto.GoodsForSale.Count == 0)

{

AddError("Список товаров пуст.");

return null;

}

foreach (var goodsUnit in dto.GoodsForSale)

{

ValidateGoodsUnit(goodsUnit);

goodsUnit.Status = GoodsStatus.AwaitingPayment;

}

if (HasErrors)

return null;

var sale = new Sale()

{

LocalSaleDate = DateTime.UtcNow,

Status = SaleStatus.YetNotPaid

};

dbAccess.CreateSale(sale, dto.GoodsForSale);

return sale.SaleId;

}

public void ValidateGoodsUnit(Goods goodsUnit)

{

if (goodsUnit.Status != GoodsStatus.InCart)

AddError("В корзину находится товар, статус которого не \"В корзине\"");

if (goodsUnit.SoftDeleted)

AddError("В корзину добавлен удаленный товар");

if (goodsUnit.Delivery?.LocalActualDeliveryDate is null || goodsUnit.ReceiptDate is null)

AddError("В корзину добавлен непоступивший на склад товар");

if (goodsUnit.Price <= 0)

AddError("В корзину добавлен товар с некорректной ценой. Цена не может быть меньше или равна 0");

// if goods unit has a sale, it must be returned. If goods unit doesn’t have sales, this loop won't be executed

foreach (var sale in goodsUnit.Sales)

if (sale.Status != SaleStatus.Returned)

AddError("В корзину добавлен товар, который входит в другую продажу и не может быть продан");

}

Метод ClearCart из листинга 1.1 является оберткой – он инкапсулирует способ очистки корзины, взаимодействие с которой сейчас реализовано через абстрактный класс CartViewerBaseController, который наследует от класса Controller (листинг 1.4). Данная реализация хранит в сессии строку с GUID всех единиц товара, находящихся сейчас в корзине, а так же их виды (GUID и вид разделены одним сепаратором, а их пары друг от друга – другим). В дальнейшем при модернизации корзины (например, добавление порядка, в котором товар был добавлен) необходимо будет изменить только класс CartViewerBaseController один раз и все его вызовы будут очищать корзину другим способом.

Взаимодействие (на самом деле, только его часть) с корзиной реализовано через наследование от абстрактный контроллер, а не через сервисный слой, так как ~~я не нашел другого способа~~  оно требует доступа к текущей сессии, что возможно только в слое приложения, а сервисный слой находится ниже слоя приложения, соответственно он «не знает» ни о классе HttpContext, ни об интерфейсе ISession, и, соответственно, он не может обрабатывать объекты их типов.

Листинг 1.4

public abstract class CartViewerBaseController : Controller

{

public string? GoodsIdsAndKindsInCart => HttpContext.Session.GetString(CommonNames.GoodsIdsInCartKey);

public string[] GoodsIdsAndKinds => GoodsIdsAndKindsInCart?.Split(CommonNames.GoodsIdSeparator, StringSplitOptions.RemoveEmptyEntries) ?? [];

public bool IsInCart(Guid goodsId) => GoodsIdsAndKindsInCart != null && GoodsIdsAndKinds!.Any(s => s.Contains(goodsId.ToString()));

public void SetNewCartValue(string newValue) => HttpContext.Session.SetString(CommonNames.GoodsIdsInCartKey, newValue);

public void ClearCart()

=> SetNewCartValue(string.Empty);

}

При успешной оплате сервер выполняет часть кода из листинга 1.5. Метод RegisterSaleAsSold (досл. зарегистрировать продажу как проданную) обрабатывает POST-запрос, который предполагается для отправки через AJAX, следующим образом: сначала производится попытка регистрации продажи с помощью сервиса saleService, вызываемого из IoC-контейнера с помощью атрибута [FromServices]. При любом исходе рассматриваемый метод возвращает текстовую информацию о результате HTTP-запроса: успех или неудача. В качестве альтернативы можно возвращать значение типа bool. При неудаче выполняется восстановление товара обратно в корзину, так как на данный момент в БД была сформирована новая продажа, включающая в себя весь товар из корзины, а из корзины этот товар был удален. Данный метод – первая ступень (слой) обработки HTTP-запроса, которая «направляет» полученные данные в следующий слой – сервисный (в данном случае представлен параметрами метода saleService и cartService).

Листинг 1.5

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<ContentResult> RegisterSaleAsSold(Guid saleId, SalePaidBy paidBy, [FromServices] IExistingSaleManagementService saleService, [FromServices] ICartService cartService, [FromServices] IGetGoodsUnitsRelatedToSaleService goodsService, [FromServices] IUpdateGoodsStatusService goodsStatusService, [FromServices] IMapKindOfGoodsService kindOfGoodsService)

{

string result;]

try

{

await saleService.RegisterSaleAsPaid(saleId, paidBy);

var goods = await goodsService.GetOrigGoodsUnitsRelatedToSale(saleId);

foreach(var goodsUnit in goods)

{

await goodsStatusService.UpdateGoodsStatus(goodsUnit.GoodsId, await kindOfGoodsService.GetGoodsKind(goodsUnit.GoodsId), GoodsStatus.Sold);

}

result = "Successfully registered";

}

catch

{

await RestoreCart(saleId, cartService);

await saleService.CancelSale(saleId);

result = "Failed to register";

}

return Content(result);

}

Следующий этап – метод RegisterSaleAsPaid сервиса с типом IExistingSaleManagementService (листинг 1.6). Сначала в данном методе у контекста базы данных запрашивается экземпляр продажи, идентификатор которой соответствует полученному параметру saleId, затем происходит проверка: оплачена ли продажа или нет. Если продажа уже оплачена, то генерируется исключение – методу был передан некорректный аргумент. Иными словами, происходит попытка зарегистрировать уже оплаченную продажу как снова оплаченную, что с точки зрения предметной области не имеет смысла.

Если продажа не была оплачена, то она помечается как оплаченная, после чего запись в контексте БД обновляется и изменения сохраняются. Если не выполнять метод Update(), то контекст не будет проверять изменения объекта sale, а если не выполнять метода SaveChanges() (в данном случае его асинхронную версию), то контекст не выполнит запросы к БД. Также важным замечанием относительно данного метода (а также всего приложения) является отсутствие обработки исключений в сервисном слое. Объясняется это тем, что исключения обрабатываются на стоящем выше уровне – контроллерами. ~~не уверен что это правильно но пока что это неплохо работает.~~  В данном случае, например, возможна ситуация, когда запись о продаже с полученным saleId не будет найдена в БД и метод SingleAsync() сгенерирует исключение.

Листинг 1.6

public async Task RegisterSaleAsPaid(Guid saleId, SalePaidBy paidBy)

{

var sale = await context.Sales.SingleAsync(sale => sale.SaleId == saleId);

if (sale.IsPaid)

throw new ArgumentException("attempt to register an already paid sale as a paid one");

sale.IsPaid = true;

sale.PaidBy = paidBy;

sale.Status = SaleStatus.Sold;

context.Update(sale);

await context.SaveChangesAsync();

}

Следующий методы для рассмотрения – RestoreCart (листинг 1.7) и CancelSale (листинг 1.8) – выполняются при генерации исключения в методе из листинга 1.5. Первый метод использует метод MoveGoodsBackToCart (переместить товар обратно в корзину) для вызова соответствующего метода у cartService, а затем устанавливает полученный список товаров как новое значение корзины.

Листинг 1.7

private async Task RestoreCart(Guid saleId, ICartService cartService)

{

string newCartContent = await cartService.MoveGoodsBackToCart(saleId);

SetNewCartValue(newCartContent);

}

Второй метод (CancelSale) находит необходимую продажу, и затем логически удаляет ее, следовательно, отмена покупки – это логическое ее удаление. В дальнейшем можно добавить функцию восстановления товара в корзине при проблемах с оплатой. Например, клиент выбрал оплату картой, но она оказалась заблокирована – продажа удаляется, а у клиента есть наличные средства, и он меняет способ оплаты. При данной реализации кассир в таком случае вынужден заполнять корзину снова.

Листинг 1.8

public async Task CancelSale(Guid saleId)

{

var sale = await context.Sales.SingleAsync(sale => sale.SaleId == saleId);

sale.SoftDeleted = true; // надо мягко удалить записи из соединяющих таблиц НЭП

await context.SaveChangesAsync();

}

Несмотря на то, что ранее было сказано, что корзина реализована через абстрактный котроллер, у записей товара в базе данных есть статус «в корзине», которое необходимо изменять соответствующе. Данная ответственность возложена на класс CartService (реализующий ICartService). В листинге 1.9 выполняется перемещение товара обратно в корзину, что происходит при безуспешной оплате в листинге 1.7. Сначала запрашивается список всех товаров, относящихся к продаже с идентификатором saleId, затем для каждого товара устанавливается статус «в корзине». Возвращаемое значение – строка с идентификаторами товаров и типами, которая затем установится как значение корзины для текущей сессии.

Листинг 1.9

public async Task<string> MoveGoodsBackToCart(Guid saleId)

{

var goods = await goodsRelatedToSaleService.GetOrigGoodsUnitsRelatedToSale(saleId);

foreach (var goodsUnit in goods)

{

goodsUnit.Status = GoodsStatus.InStock;

context.Update(goodsUnit);

}

context.SaveChanges();

return string.Join(GoodsIdSeparator, goods.Select(goodsUnit => new { goodsUnit.GoodsId, TypeName = goodsUnit.GetType().Name }));

}

Далее, в листинге 1.10, описывается способ получения списка всех товаров с помощью методов расширения Cast() и Concat() – метод Cast приводит объекты товаров разных типов к общему классу Goods, который наследуется ими всеми, а метод Concat объединяет все четыре коллекции товаров в одну.

Листинг 1.10

public async Task<List<Goods>> GetOrigGoodsUnitsRelatedToSale(Guid saleId)

{

var sale = await saleService.GetOriginalSale(saleId);

return sale.MusicalInstruments

.Cast<Goods>()

.Concat(sale.Accessories

.Cast<Goods>())

.Concat(sale.SheetMusicEditions

.Cast<Goods>())

.Concat(sale.AudioEquipmentUnits

.Cast<Goods>())

.ToList();

}

Несмотря на то, что все описанные выше методы используются в данном предложении во множественном числе, вместе они все еще составляют алгоритм работы одного и того же действия – оформления продажи. Тем не менее, данное действие разделено на два этапа: создание неоплаченной продажи и последующее регистрирование этой продажи как оплаченной (при отсутствии ошибок). В итоге это действие рассматривается как единое целое.

1. Подробно описать один наиболее интересный алгоритм информационной системы (сложный). Алгоритмы проиллюстрировать блок-схемами. Алгоритмы обработки информации, описание связей модулей и БД и т.д.
2. В зависимости от технологии реализации представить дерево функций и сценарий диалога, диаграмму классов, схему взаимосвязи программных модулей и информационных файлов и/или другие модели, служащие для описания программного обеспечения. Описать данные модели.

# 2 Тестирование элементов системы и формирование технологической среды

1. Обоснование выбора технологии тестирования, наиболее соответствующей особенностям разработанной системы, и ее характеристика.
2. Разработать тесты для спроектированного в ПО (объяснить выбор вида тестирования, почему применяются именно эти, а не другие тестовые комбинации пути прохождения вычислительного процесса).
3. Привести результаты тестирования в виде таблиц и экранных форм. Описание тестов, результатов тестирования и их анализ. В тестировании предусмотреть проверку выполнения требований по надежности и защите информации. Результаты тестирования рекомендуется оформлять в виде таблицы, в которой указываются тестовые воздействия и соответствующие реакции ИС. При целесообразности можно добавить экранные формы для иллюстрации результатов некоторых тестов.

# 3 Анализ проблем автоматизации процесса продаж музыкального оборудования и пути их решения

# ЗАКЛЮЧЕНИЕ

Таким образом, все поставленные в работе задачи выполнены, и, соответственно, цель достигнута.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ



# **ПРИЛОЖЕНИЕ А**

Экранные формы системы автоматизации

# **ПРИЛОЖЕНИЕ Б**

Программный код элементов системы автоматизации

# ПРИЛОЖЕНИЕ В

Тест-кейсы для тестирования программного обеспечения