Картина, която съдържа текст, Шрифт, лого, символ

Описанието е генерирано автоматично

**Курсов проект по ПС Computer Service**

Тема 1: Система за търсене и показване на годишни списъци

Изготвил: Стоян Стойков  
Специалност: КСИ  
Фак. номер: 121221087  
Група: 45

22.05.2024

Съдържание

[Анализ на заданието 3](#_Toc167296877)

[Компоненти на проекта 3](#_Toc167296878)

[ER Диаграма на класовете 3](#_Toc167296879)

[Файлова стриктура 3](#_Toc167296880)

[App.xaml 4](#_Toc167296881)

[Database folder 4](#_Toc167296882)

[DatabaseContext.cs 4](#_Toc167296883)

[DatabaseService.cs 5](#_Toc167296884)

[Helpers folder 6](#_Toc167296885)

[DelegateCommands.cs 6](#_Toc167296886)

[Model Folder 7](#_Toc167296887)

[Computer.cs 7](#_Toc167296888)

[Repair.cs 7](#_Toc167296889)

[View folder 8](#_Toc167296890)

[SearchControl View 8](#_Toc167296891)

[AddRepairWindow 8](#_Toc167296892)

[MainWindow 9](#_Toc167296893)

[ViewModels folder 9](#_Toc167296894)

[SearchViewModel 9](#_Toc167296895)

[AddRepairViewModel 10](#_Toc167296896)

# Анализ на заданието

Приложението е направено за сервиз за компютри, с което да следи направените ремонти. Базирано на WPF. Позволява на потребителите да търсят ремонти въз основа на специфични критерии и да добавят нови ремонти, ако не съществуват в базата данни. Приложението следва модела на проектиране MVVM (Model-View-ViewModel), за да поддържа чисто разделение между потребителския интерфейс и бизнес логиката.

# Компоненти на проекта

## ER Диаграма на класовете

## Файлова стриктура

ComputerServiceApplication/

│

├── Computer\_Serivce/

│ ├── Database/

│ │ ├── DatabaseContext.cs

│ │ ├── DatabaseService.cs

│ │

│ ├── Helpers/

│ │ ├── DelegateCommand.cs

│ │

│ ├── Model/

│ │ ├── Computer.cs

│ │ └── Repair.cs

│ │

│ ├── View/

│ │ ├── Controls/

│ │ │ ├── SearchControl.xaml

│ │ │ └── SearchControl.xaml.cs

│ │ ├── Window/

│ │ │ ├── AddRepairWindow.xaml

│ │ │ └── AddRepairWindow.xaml.cs

│ │ │ ├── MainWindow.xaml

│ │ │ └── MainWindow.xaml.cs

│ │

│ ├── ViewModel/

│ │ ├── AddRepairViewModel.cs

│ │ └── SearchViewModel.cs

│ │

│ └── App.xaml

│ └── App.xaml.cs

│

└── README.md

## Функционални изисквания

* **Търсене и филтриране:** Потребителят трябва да може да задава критерии за търсене (филтри) за полетата на обектите и да е възможно търсенето по конкретна година и връщане на резултати за тази година
* **Алтернативни резултати:** Ако няма резултати за годината, приложението трябва да връща резултати от последната налична година, за която има данни, отговарящи на останалите критерии.
* **Спазване на MVVM**: Архитектурата на системата трябва да спазва MVVM модела.

## App.xaml

В проекта Computer\_Serivce всички стилове за цялото приложение са дефинирани във файла App.xaml. Този централизиран подход гарантира, че стиловете се прилагат в цялото приложение. Дефинира ресурсите, стиловете и настройките за цялото приложение.

## Database folder

Папката Database съдържа класовете и конфигурациите, необходими за взаимодействие с базата данни. Това включва настройка на контекста на базата данни, дефиниране на начина, по който обектите се съхраняват и извличат, и предоставяне на методи за заявки и актуализиране на базата данни. Това се случва в два файла DatabaseContext.cs и DatabaseService.cs.

### DatabaseContext.cs

Класът DatabaseContext наследява от DbContext и отговаря за конфигурирането на връзката към базата данни и настройването на моделите на обекти. И накрая се добавят няколко обекта при първото създаване на таблиците.

public class DatabaseContext : DbContext

{

public DbSet<Repair> Repairs { get; set; }

public DbSet<Computer> Computers { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

string baseDir = AppDomain.CurrentDomain.BaseDirectory;

string databaseFile = "ComputerRepairs.db";

string projectRoot = Path.GetFullPath(Path.Combine(baseDir, @"..\..\..\.."));

string databaseDir = Path.Combine(projectRoot, "db");

if (!Directory.Exists(databaseDir))

{

Directory.CreateDirectory(databaseDir);

}

string databasePath = databaseDir + $"\\{databaseFile}";

optionsBuilder.UseSqlite($"Data Source={databasePath}")

.LogTo(Console.WriteLine);

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Repair>().Property(e => e.Id).ValueGeneratedOnAdd();

modelBuilder.Entity<Computer>().Property(e => e.Id).ValueGeneratedOnAdd();

base.OnModelCreating(modelBuilder);

// Configure one-to-many relationship

modelBuilder.Entity<Repair>()

.HasOne(r => r.Computer)

.WithMany(v => v.Repairs)

.HasForeignKey("ComputerId");

modelBuilder.Entity<Repair>().Property(e => e.Id).ValueGeneratedOnAdd();

modelBuilder.Entity<Computer>().Property(e => e.Id).ValueGeneratedOnAdd();

// Seed data for Computer

modelBuilder.Entity<Computer>().HasData(

new Computer { Id = 1, Brand = "Lenovo", Model = "Yoga X1", YearMade = "2018", SerialNumber = "LE2018y" },

// ...

);

// Seed data for Repair

modelBuilder.Entity<Repair>().HasData(

new Repair { Id = 1, ComputerId = 1, YearOfService = 2023, ServiceType = "Hardware Maintenance", Description = "Full system diagnostics and cleaning" },

// ...

);

}

### DatabaseService.cs

Класът DatabaseService предоставя методи за взаимодействие с базата данни. Той използва DatabaseContext за извършване на CRUD операции (Създаване, Четене, Актуализиране, Изтриване). Най-забележителният метод в този клас е общият метод за търсене Search<T>.

**Search<T> метод**

Методът Search<T> е общ метод, който позволява да търсите обекти от произволен тип (T) въз основа на набор от условия (филтри), конкретна година и незадължителни включвания за свързани обекти (include). В този проект се използва, за да покаже свързаните към Repairs компютри.

**Параметри:**

Параметри:

* List<Expression<Func<T, bool>>> filters: Списък от ламбда изрази, представящи условията, на които трябва да отговарят обектите.
* Expression<Func<T, int>> yearExpression: Ламбда израз за извличане на свойството година от обекта.
* int? searchYear: незадължителна година за търсене на обекти от тази година или по-рано.
* Func<IQueryable<T>, IIncludableQueryable<T, object>>? includeExpression: Незадължителен ламбда израз за включване на свързани обекти в заявката.

public IEnumerable<T> Search<T>(

List<Expression<Func<T, bool>>> filters,

Expression<Func<T, int>> yearExpression,

int? searchYear,

Func<IQueryable<T>, IIncludableQueryable<T, object>>? includesExression = null

) where T : class

{

using (var context = new DatabaseContext())

{

var query = context.Set<T>().AsQueryable();

// Apply includes if provided

if (includesExression != null)

{

query = includesExression(query);

}

// Apply filters to the query

foreach (var filter in filters)

{

query = query.Where(filter);

}

// Search for entities based on the provided year

if (searchYear.HasValue)

{

var yearQuery = query.Select(yearExpression);

var closestYear = yearQuery.Where(y => y <= searchYear.Value).OrderByDescending(y => y).FirstOrDefault();

if(closestYear == 0)

{

return Enumerable.Empty<T>();

}

var parametar = yearExpression.Parameters[0];

var equalsClosestYear = Expression.Equal(yearExpression.Body, Expression.Constant(closestYear, typeof(int)));

var lambda = Expression.Lambda<Func<T, bool>>(equalsClosestYear, parametar);

query = query.Where(lambda);

}

return query.ToList();

}

}

## Helpers folder

### DelegateCommands.cs

Класът DelegateCommand е реализация на интерфейса ICommand. Използва се за обработка на командната логика във ViewModel, позволявайки на ViewModel да реагира на потребителските взаимодействия. Този клас е от съществено значение за обвързване на команди в XAML, което отделя потребителския интерфейс от бизнес логиката.

**Основни методи:**

1. Конструктор: Конструкторът DelegateCommand приема два параметъра:
   * execute: действие, което определя какво се случва, когато командата се изпълни.
   * canExecute: Функция, която връща булево значение, показващо дали командата може да бъде изпълнена. Този параметър не е задължителен.
2. CanExecute: Този метод проверява дали командата може да бъде изпълнена. Ако е предоставена функцията canExecute, тя връща своя резултат. В противен случай връща true, което показва, че командата винаги може да се изпълни.
3. Execute: Този метод изпълнява действието, което е подадено.
4. RaiseCanExecuteChanged: Този метод задейства събитието CanExecuteChanged, което уведомява WPF, че възможността за изпълнение на командата може да се е променила. Това е важно за активиране или деактивиране на бутони или други източници на команди в потребителския интерфейс.
5. CanExecuteChanged event: Това събитие е част от интерфейса на ICommand и се използва от WPF за проследяване на промени в състоянието на изпълнение на командата. DelegateCommand се връзва за събитието CommandManager. RequerySuggested за автоматично преоценяване на метода CanExecute при определени условия (напр. когато фокусът на потребителския интерфейс се промени).

## Model Folder

### Computer.cs

Компютърният клас представлява компютърен обект с различни свойства, описващи неговите атрибути. Той също така поддържа връзка с класа Repair за проследяване на ремонтите, свързани с всеки компютър. Класът Computer има връзка "един към много" с класа Repair, което означава, че един компютър може да има няколко ремонта. Това е представено от ICollection<Repair> в класа Computer и свойството Computer в класа Repair.

Атрибути:

* Id: Първичният ключ за компютърния обект. Той се генерира автоматично от базата данни.
* Brand: Марката на компютъра (напр. Dell, HP, Apple). Това поле е задължително.
* Model: Моделът на компютъра (напр. XPS 13, MacBook Pro). Това поле е задължително.
* YearMade: Годината на производство на компютъра. Това поле не е задължително.
* SerialNumber: Серийният номер на компютъра, който го идентифицира уникално. Това поле е задължително.
* Repairs: Колекция от поправки, свързани с компютъра.

### Repair.cs

Класът Repair представлява запис за ремонт на компютър. Той включва подробности за услугата за ремонт и поддържа препратка към свързания компютър.

Атрибути:

* Id: Първичният ключ за обекта за поправка. Той се генерира автоматично от базата данни.
* YearOfService: Годината на извършване на ремонтната услуга. Това поле е задължително.
* ServiceType: Типът извършена услуга (напр. ремонт на хардуер, актуализация на софтуера). Това поле е задължително.
* Description: Подробно описание на ремонтната услуга. Това поле не е задължително.
* ComputerId: Външният ключ, който свързва поправката с конкретен компютър.
* Компютър: Свойство за навигация към асоциирания компютърен обект.

## View folder

### SearchControl View

Изгледът SearchControl е потребителски контрол в проекта Computer\_Serivce, който предоставя потребителски интерфейс за търсене на записи за ремонт на компютри въз основа на специфични критерии като година на обслужване, марка и модел.

XAML файлът дефинира оформлението и външния вид на потребителския интерфейс SearchControl. Той включва различни елементи на потребителския интерфейс, като етикети, текстови полета, бутони и мрежа с данни.

Ключови раздели:

* Grid Layout: Организира елементите на потребителския интерфейс в редове и колони.
* TextBlocks: Показване на заглавия и текст.
* StackPanels: Групирайте свързани полета за въвеждане и бутони.
* TextBoxes: Позволява потребителско въвеждане за критерии за търсене.
* Buttons: Задействайте търсене и ясни действия.
* DataGrid: Показва резултатите от търсенето.

Изгледът SearchControl обвързва своите UI елементи със свойства и команди, дефинирани в SearchViewModel. Това обвързване позволява динамични актуализации и потребителски взаимодействия, без да се изисква директно манипулиране на UI елементи в задния код.

Ключови връзки:

* Text Boxes: Свързване към свойствата RepairedYear, Brand и Model в SearchViewModel.
* Buttons: Свързване към SearchRepairsCommand и ClearCommand в SearchViewModel.
* DataGrid: Свързва се с колекцията Repairs

Обработката на събития се извършва в code-behind-а за специфични случаи, като например валидиране на въведената година от потребителя. Събитието е NumberValidationTextBox - гарантира, че в текстовото поле за годината е разрешено само въвеждане на цифри.

### AddRepairWindow

AddRepairWindow е изскачащ прозорец в проекта, който позволява на потребителите да добавят нови записи за ремонт за компютър, ако по търсените критерии няма такъв. Този прозорец предоставя удобен за потребителя интерфейс за въвеждане на ремонт и след това за запис в базата данни.

XAML файлът дефинира оформлението и външния вид на AddRepairWindow. Той включва етикети, текстови полета за въвеждане и бутон за добавяне на поправката.

Изгледът AddRepairWindow свързва своите UI елементи със свойства, дефинирани в AddRepairViewModel. Това свърззване позволява динамични актуализации и потребителски взаимодействия, без да се изисква директно манипулиране на UI елементи в задния код. Обработката на събития се управлява чрез кода за специфични случаи, като валидиране на въведените от потребителя данни за година, да е възможно въвеждането само на цифри и обработка на щракването върху бутона за добавяне. Не е направено с команда, защото имаше проблем при затваряне на изкачващия прозорец.

Ключови връзки:

* Текстови полета: Свързване към свойствата SerialNumber, RepairedYear, Brand, Model и Description в AddRepairViewModel.
* Бутон: Свързва се с AddRepairCommand в AddRepairViewModel.

### MainWindow

Това е основният прозорец, който обединява SearchControl-а и изкачващия прозорец за добавяне на поправка.

## ViewModels folder

### SearchViewModel

SearchViewModel е отговорен за управлението на логиката за търсене и показване на компютърни ремонти. Той обработва въведените от потребителя критерии за търсене, извършва търсения и управлява резултатите.

**Properties:**

* Computers: ObservableCollection<Computer>, която съдържа списъка с компютри.
* Repairs: ObservableCollection<Repair>, която съдържа резултатите от търсенето за поправки.
* RepairedYear, Brand, Model: низове, които съдържат въведеното от потребителя за критерии за търсене.

**Commands:**

* SearchRepairsCommand: Команда за изпълнение на търсенето въз основа на въвеждане от потребителя.
* ClearCommand: Команда за изчистване на критериите за търсене и резултатите.

**Ключови методи:**

* LoadComputers: Зарежда всички компютри от базата данни и попълва колекцията Computers.
* LoadRepairs: Зарежда всички поправки от базата данни и попълва колекцията Repairs.
* SearchRepairs: Търси ремонти въз основа на критериите, предоставени от потребителя, и актуализира колекцията Repairs. Ако не бъдат намерени резултати, той подканва потребителя да добави нова поправка.
* ClearFields: Изчиства критериите за търсене и презарежда всички поправки.

### AddRepairViewModel

AddRepairViewModel е отговорен за управлението на логиката за добавяне на нов ремонт. Той обработва въведените от потребителя данни за новия ремонт и записва новия ремонт в базата данни.

**Properties:**

* Brand, Model, Description, RepairedYear, SerialNumber, YearMade: Низове, които съдържат въведените от потребителя данни за новия ремонт.

**Команди:**

* AddRepairCommand: Команда за изпълнение на логиката за добавяне на нова поправка.

**Ключови методи:**

* AddRepair: Създава нов обект Repair и Computer въз основа на въведеното от потребителя и ги записва в базата данни.
* ClearProps: Изчиства всички входни свойства за нулиране на формуляра.