Вопрос 1:

Паттерн проектирования - это повторяемый подход или решение для часто встречающихся проблем в разработке программного обеспечения. Он описывает как решать конкретную проблему с помощью определенного набора правил и шаблонов.

Некоторые примеры архитектурных паттернов в программном обеспечении:

1. MVC (Model-View-Controller) - разделение приложения на три компонента для лучшей организации кода и удобства его поддержки.

2. Singleton - гарантирование, что класс имеет только один экземпляр и обеспечивает глобальный доступ к этому экземпляру.

3. Factory Method - создание объектов, не указывая конкретный класс создаваемого объекта.

4. Observer - обеспечение возможности подписки на изменения, происходящие в объекте.

5. Decorator - динамическое добавление функциональности объекту без изменения его исходного кода.

Вопрос 2:

MVVM (Model-View-ViewModel) - архитектурный паттерн программирования, который разделяет приложение на три компонента: модель, представление и модель представления. Модель представления (ViewModel) представляет интерфейс между моделью и представлением (View) и содержит бизнес-логику для управления данными и поведением представления. Модель (Model) представляет собой данные и бизнес-логику приложения, а представление - пользовательский интерфейс, с которым пользователь взаимодействует. В своей основе паттерн MVVM направлен на обеспечение лучшей разделенности слоев приложения и более четкой организации кода.

Вопрос 3:

В паттерне MVVM (Model-View-ViewModel) View отвечает за представление данных и взаимодействие с пользователем. В конкретном случае, View может отображать данные, полученные из ViewModel, и реагировать на действия пользователя, например, нажатие кнопок или ввод текста.

В WPF (Windows Presentation Foundation) View может быть представлен как XAML-файл, который определяет графический интерфейс и размещение элементов управления на окне. View может также быть создан программно в коде C#, используя API WPF.

Вопрос 4:

ViewModels в MVVM отвечает за представление данных и управление состоянием представления. Он предоставляет данные и повышает уровень абстракции между представлением и моделью. ViewModel также отвечает за преобразование данных, необходимых для отображения на экране.

Отделение бизнес-логики от логики представления важно, потому что это позволяет создать приложение, которое более гибкое и легко поддерживаемое. Если бизнес-логика связана с представлением, ее будет сложно модифицировать, так как бизнес-логика усложнится и станет уязвимой к ошибкам. Отделение бизнес-логики от логики представления позволяет разработчикам ясно видеть, какие данные используются в представлении, и как происходит взаимодействие между компонентами приложения, что повышает читабельность и снижает количество ошибок в коде.

Вопрос 5:

В архитектуре MVVM (Model-View-ViewModel) Models отвечает за представление данных в приложении. Это слой, который обращается к источникам данных, таким как базы данных или серверы, и получает данные, необходимые для отображения в пользовательском интерфейсе.

Пример инструмента, который обеспечивает удобный доступ к данным, - это ORM-фреймворки (Object-Relational Mapping). Они позволяют программистам работать с базами данных, используя объектно-ориентированный подход, что делает код более читабельным и легко поддерживаемым. Например, Entity Framework в .NET и Hibernate в Java - это популярные ORM-фреймворки. Они предоставляют удобный API для работы с базами данных и могут автоматически создавать SQL-запросы на основе моделей данных.

Вопрос 6:

Scaffold-DbContext "Server=(local);Database=TestDatabase;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models

Вопрос 7:

Интерфейс INotifyPropertyChanged в .NET используется для уведомления об изменении значений свойств объекта. Классы, которые реализуют этот интерфейс, могут отправлять событие PropertyChanged всем подписчикам, чтобы сообщить им об изменении значения одного из своих свойств.

Это особенно полезно при создании пользовательских интерфейсов, когда данные в модели могут изменяться динамически, а представление должно отображать эти изменения для пользователя. Использование INotifyPropertyChanged обеспечивает более эффективный и оптимизированный механизм передачи информации о том, что свойство изменилось и какое новое значение установлено.

Вопрос 8:

ObservableCollection<T> - это коллекция элементов, которая реализует интерфейс INotifyCollectionChanged. Это означает, что при добавлении, удалении или изменении элементов в коллекции, коллекция автоматически генерирует событие CollectionChanged. Для привязки этой коллекции к пользовательскому интерфейсу, такой как ListBox или DataGrid, это обеспечивает автоматическое обновление представления при изменении коллекции.

Обычные коллекции, например List<T>, не реализуют интерфейс INotifyCollectionChanged и не генерируют событий при изменении элементов. Хотя вы можете использовать обычную коллекцию для хранения данных, но если вы хотите, чтобы представление автоматически обновлялось при изменении коллекции, вы должны вручную обновлять представление.

Еще одно отличие между ObservableCollection<T> и List<T> заключается в том, что первый предоставляет дополнительный функционал, который позволяет легко добавлять и удалять несколько элементов за одну операцию, используя методы AddRange() и RemoveRange(). Этот функционал отсутствует в стандартной коллекции List<T>.

Кроме того, следует отметить, что ObservableCollection<T> имеет некоторые особенности, важные для работы с многопоточностью. Для того чтобы обойти проблему доступа к коллекции из нескольких потоков одновременно, ObservableCollection<T> следует использовать только из главного (UI) потока. Если вы хотите изменять коллекцию из других потоков, то нужно использовать метод Dispatcher.Invoke() или использовать другие подходящие механизмы синхронизации.

Вопрос 9:

Свойство DataContext в WPF (Windows Presentation Foundation) является одним из ключевых механизмов связывания данных и позволяет установить объект, который будет использоваться в качестве источника данных для элементов управления в окне или странице.

Вопрос 10:

Команда в WPF - это объект, который представляет действие, которое может быть выполнено пользователем через элементы управления, такие как кнопки, меню и контекстные меню. Команда позволяет вынести логику выполнения действия из элемента управления в отдельный объект, что обеспечивает более гибкую и масштабируемую архитектуру приложения.

Интерфейс ICommand определяет следующие методы:

Execute(object parameter): метод, который вызывается при выполнении команды. В качестве параметра передается объект, который может использоваться для передачи дополнительной информации, например, выбранного элемента списка или значения текстового поля.

CanExecute(object parameter): метод, который определяет, может ли команда выполняться в текущем состоянии приложения. Если метод возвращает true, то команда доступна для выполнения, если false - то не доступна. Также возможно указание условий, при которых команда может быть доступна, например, когда выбран элемент в списке.

Метод Execute используется для выполнения основной логики команды, например, сохранения данных или открытия нового окна. Метод CanExecute используется для проверки условий, при которых команда может быть доступна или недоступна для выполнения. Это может быть полезно, например, для блокировки кнопки "Сохранить", если форма не заполнена полностью.

Классы, реализующие интерфейс ICommand, могут быть созданы вручную или использоваться готовые классы, такие как RelayCommand из пакета MVVM Light Toolkit или DelegateCommand из пакета Prism. Эти классы облегчают создание команд и предоставляют дополнительный функционал, такой как автоматическое обновление состояния команды при изменении свойств модели представления.

Вопрос 11:

Пример реализации класса RelayCommand из MVVM Light Toolkit:

public class RelayCommand : ICommand

{

private readonly Action<object> \_execute;

private readonly Predicate<object> \_canExecute;

public RelayCommand(Action<object> execute, Predicate<object> canExecute = null)

{

\_execute = execute ?? throw new ArgumentNullException(nameof(execute));

\_canExecute = canExecute;

}

public event EventHandler CanExecuteChanged

{

add => CommandManager.RequerySuggested += value;

remove => CommandManager.RequerySuggested -= value;

}

public bool CanExecute(object parameter) => \_canExecute == null || \_canExecute(parameter);

public void Execute(object parameter) => \_execute(parameter);

}

Пример использования команды на стороне модели представления:

csharpCopy Code

private RelayCommand \_saveCommand;

public ICommand SaveCommand => \_saveCommand ??= new RelayCommand(Save, CanSave);

private void Save(object parameter)

{

*// Логика сохранения данных*

}

private bool CanSave(object parameter)

{

return !string.IsNullOrEmpty(Name) && !string.IsNullOrEmpty(Email);

}

В данном примере мы создали команду SaveCommand, которая вызывает метод Save при выполнении и метод CanSave для проверки доступности команды. В методе Save мы можем выполнять логику сохранения данных, а в методе CanSave определять, может ли команда быть выполнена в текущем состоянии.

Пример привязки команды к кнопке в XAML:

Copy Code

<Button Content="Save" Command="{Binding SaveCommand}" />

В этом примере мы создали кнопку с надписью "Save" и привязали ее к команде SaveCommand, которая определена в модели представления. Теперь при нажатии на кнопку будет выполняться метод Save из модели представления. Кроме того, кнопка автоматически будет блокироваться, если метод CanSave вернет false.

Вопрос 12:

Для произведения валидации в рамках паттерна MVVM можно использовать два способа:

Использование ValidationRule - это класс, который наследуется от абстрактного класса ValidationRule и позволяет проверять входные данные, возвращая ValidationResult, содержащий информацию об ошибке (если она есть). Пример использования ValidationRule для проверки количества товара может выглядеть следующим образом:

public class QuantityValidationRule : ValidationRule { public override ValidationResult Validate(object value, CultureInfo cultureInfo) { int.TryParse(value.ToString(), out int quantity); if (quantity <= 0) { return new ValidationResult(false, "Количество товара должно быть положительным числом."); } return ValidationResult.ValidResult; } }

Использование интерфейса IDataErrorInfo - это интерфейс, который определяет два метода: Error и Item[]. Метод Error возвращает строку с сообщением об ошибке, а метод Item[] возвращает сообщение об ошибке для заданного свойства. Пример использования IDataErrorInfo для проверки сложности пароля может выглядеть следующим образом:

public class User : IDataErrorInfo { public string Password { get; set; } public string this[string propertyName] { get { string error = null; if (propertyName == "Password") { if (!Regex.IsMatch(Password, @"^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d)[a-zA-Z\d]{8,}$")) { error = "Пароль должен содержать буквы в верхнем и нижнем регистре и хотя бы одну цифру; длина пароля 8 или более знаков."; } } return error; } } public string Error { get { return null; } } }

Вопрос 13:

Для произведения валидации в рамках паттерна MVVM можно использовать два способа:

Использование ValidationRule - это класс, который наследуется от абстрактного класса ValidationRule и позволяет проверять входные данные, возвращая ValidationResult, содержащий информацию об ошибке (если она есть). Пример использования ValidationRule для проверки количества товара может выглядеть следующим образом:

public class QuantityValidationRule : ValidationRule { public override ValidationResult Validate(object value, CultureInfo cultureInfo) { int.TryParse(value.ToString(), out int quantity); if (quantity <= 0) { return new ValidationResult(false, "Количество товара должно быть положительным числом."); } return ValidationResult.ValidResult; } }

Использование интерфейса IDataErrorInfo - это интерфейс, который определяет два метода: Error и Item[]. Метод Error возвращает строку с сообщением об ошибке, а метод Item[] возвращает сообщение об ошибке для заданного свойства. Пример использования IDataErrorInfo для проверки сложности пароля может выглядеть следующим образом:

public class User : IDataErrorInfo { public string Password { get; set; } public string this[string propertyName] { get { string error = null; if (propertyName == "Password") { if (!Regex.IsMatch(Password, @"^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d)[a-zA-Z\d]{8,}$")) { error = "Пароль должен содержать буквы в верхнем и нижнем регистре и хотя бы одну цифру; длина пароля 8 или более знаков."; } } return error; } } public string Error { get { return null; } } }