ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ

ФАКУЛТЕТ КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ

СПЕЦИАЛНОСТ КОМПЮТЪРНО И СОФТУЕРНО ИНЖЕНЕРСТВО

Студент: Валери Ивайлов Райков

Фак. №: 121222139

Група: 42Б

Софтуерни шаблони

Курсов проект по Задание №9

Дата на предаване: 22/11/2025г.

Съдържание

Въведение в проекта

Wedding Seating Planner е интелигентна софтуерна система, предназначена да улесни и автоматизира процеса на разпределяне на гости по маси по време на сватбени мероприятия. Приложението комбинира усъвършенствани софтуерни архитектурни шаблони с интуитивен графичен интерфейс, за да предостави на организаторите мощен инструмент за справяне с един от най-сложните аспекти на сватбеното планиране. В следващите секции ще разгледаме по-детайлно кои архитектурни шаблони са използвани, как са имплементирани в конкретното приложение и ще представим графично връзката между отделните класове.

Софтуерни Предпоставки

1. .NET Framework / .NET Runtime
2. Windows 10/11 операционна система
3. Visual Studio 2022
4. Git – за изтегляне на кода

Какво да очаква потребителя

Основно управление:

- Управление на гости (добавяне, автоматично групиране и общ преглед)
- Организация на маси (създаване, проследяване, визуален преглед на разпределението и информация за всяка маса)
- Интелигентни бизнес правила (автоматична валидация, ограничения и система са нотификация и проследяване)

Допълнитени характеристики:

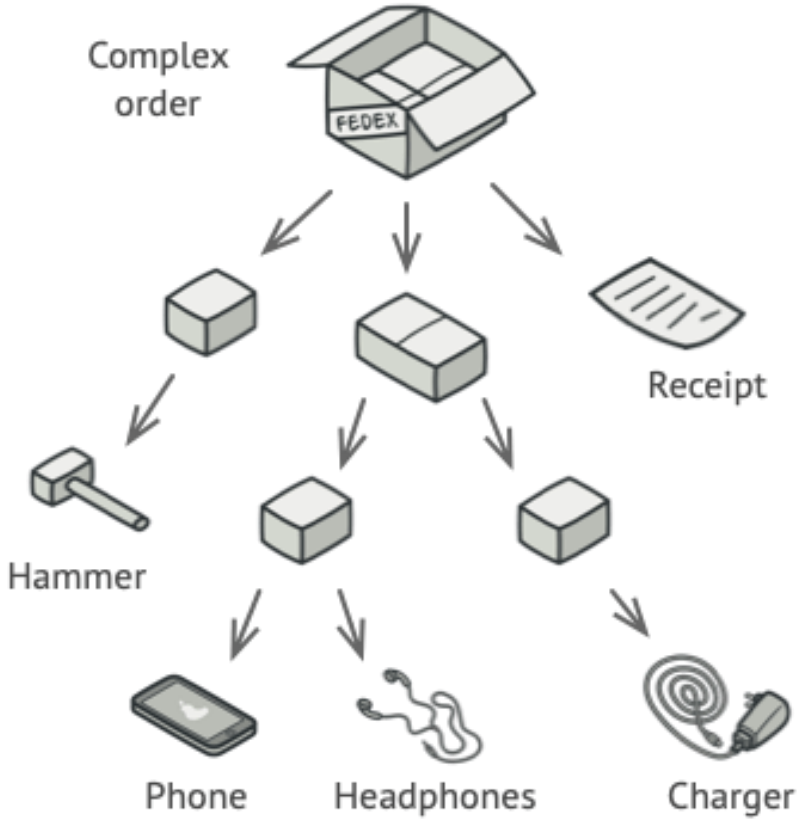
- Интуитивен графичен интерфейс (лява част: контроли, дясна част: данни)
- Обратна връзка в реално-време (актуализиране на интерфейса, визуални индикатори)
- Автоматична валидация и защита (предотвратяване на грешки, ясни съобщения, защита срещу конфликти между семейства)

Предимства на този софтуер

- Спестява време от калкулации и валидация на правила
- Избягва грешки благодарение на автоматичните валидации
- Професионални резултати с организирано и балансирано разпределение

Архитектура и Design Patterns

1. Composite Pattern



Цел и предназначение:

Composite Pattern е структурен шаблон за проектиране, който се използва за създаване на йерархична структура от обекти, които могат да бъдат третирани по един и същ начин. В контекста на приложението, това позволява да се управляват гости, семейства и маси като единна структура.

Имплементация:

```
namespace WeddingPlannerWPF.Models.Interfaces
{
    public interface ISeatComponent
    {
        public string Name { get; }
        public int GetGuestCount();
        public IEnumerable<Guest> GetGuests();
```

```
        protected bool CanAdd(ISeatComponent component);
        protected void Add(ISeatComponent component);
        protected void Remove(ISeatComponent component);
        public List<string> GetFamilies();
    }
}
```

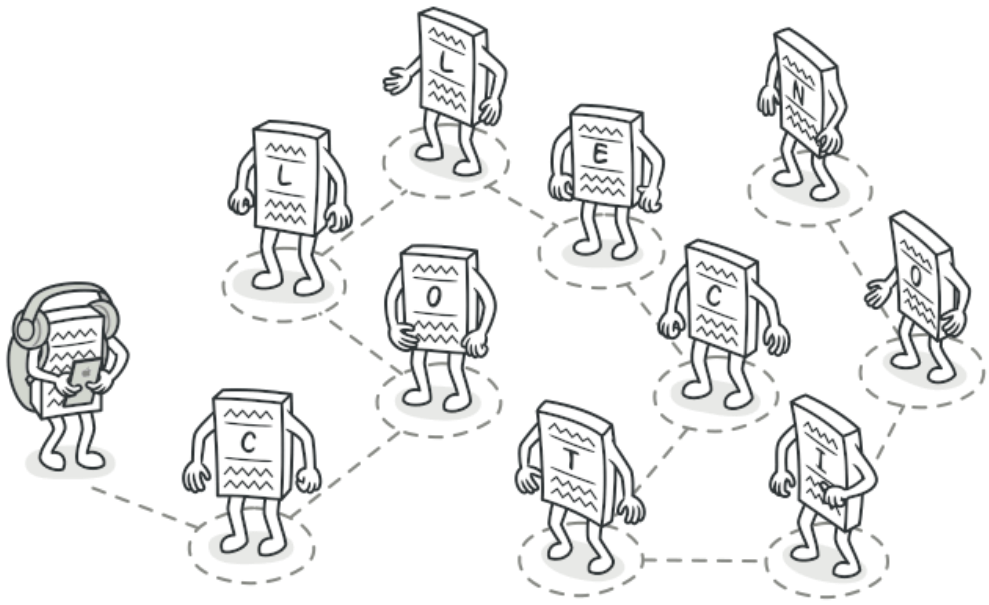
Йерархия на компонентите:

- **Guest (Leaf)**: Представлява отделен гост
- **Family (Composite)**: Представлява семейство, съдържащо гости
- **Table (Composite)**: Представлява маса, съдържаща семейства

Предимства в контекста на приложението:

- **Еднообразно третиране:** Всички компоненти се третират по един и същ начин
- **Рекурсивни операции:** Операции като броене на гости работят рекурсивно
- **Гъвкава структура:** Лесно добавяне на нови типове компоненти
- **Поддръжка на бизнес правила:** Лесно прилагане на правила за разпределение

2. Iterator Pattern



Цел и предназначение:

Iterator Pattern е поведенчески шаблон за проектиране, който предоставя стандартен начин за последователно обхождане на елементите на колекция, без да се разкрива нейната вътрешна структура. Той позволява на различни типове колекции (списъци, масиви, дървета) да бъдат обхождани по един и същ начин, използвайки единен интерфейс за итерация.

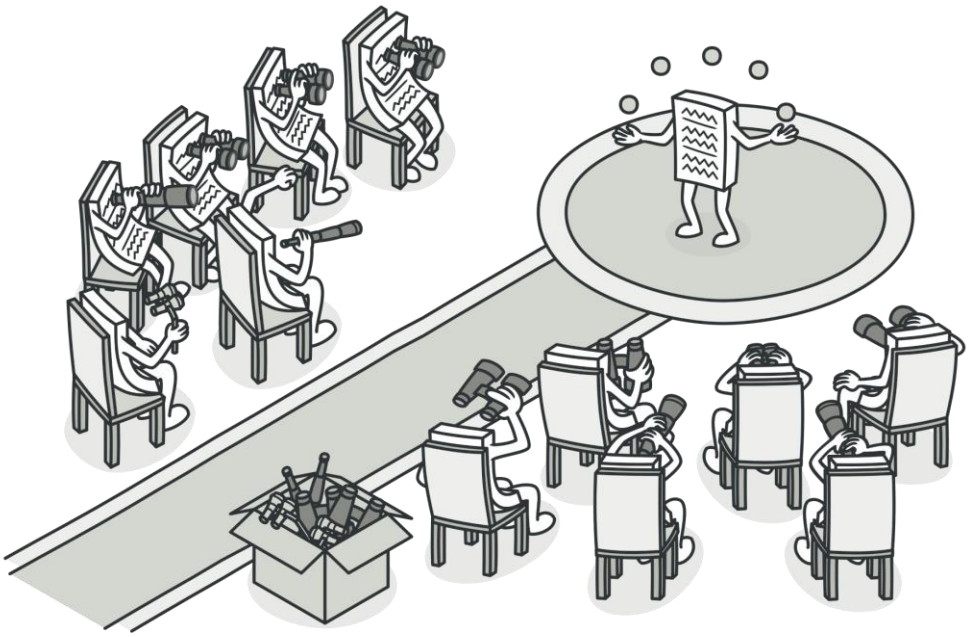
Имплементация:

```
// B Guest (Leaf)
public IEnumerable<Guest> GetGuests()
{
    yield return this;
}
// B Family (Composite)
public IEnumerable<Guest> GetGuests()
{
    foreach (var guest in Members)
    {
        yield return guest;
    }
}
// B Table (Composite)
public IEnumerable<Guest> GetGuests()
{
    foreach (var component in Components)
    {
        foreach (var guest in component.GetGuests())
        {
            yield return guest;
        }
    }
}
```

Предимства в контекста на приложението:

- **Унифициран достъп:** Единен интерфейс за обхождане на гости
- **Скриване на имплементацията:** Клиентският код не знае за вътрешната структура
- **Поддръжка на сложни структури:** Рекурсивно обхождане на йерархични структури

3. Observer Pattern (Допълнителен шаблон)



Цел и предназначение:

Observer Pattern е поведенчески шаблон за проектиране, който дефинира зависимост "един-към-много" между обекти, така че когато един обект промени състоянието си, всички зависещи от него обекти се уведомяват и актуализират автоматично.

Имплементация:

```
// Това е observer class (класа наблюдател)

public interface IweddingObserver

{

    void OnGuestAdded(Guest guest, Table table);

    void OnGuestRemoved(Guest guest, Table table);

    void OnFamilyBanned(string family1, string family2, Table
table);
```

```
    void OnRuleViolation(string message);
}

// Това е Subject class (класа който бива наблюдаван)
public interface IWeddingSubject
{
    void Attach(IWeddingObserver observer);
    void Detach(IWeddingObserver observer);
    void NotifyGuestAdded(Guest guest, Table table);
    void NotifyGuestRemoved(Guest guest, Table table);
    void NotifyFamilyBanned(string family1, string family2, Table
table);
    void NotifyRuleViolation(string message);
}
```

Предимства в контекста на приложението:

- **Автоматично уведомяване:** Следене на промени в разпределението в реално време
- **Разделяне на отговорности:** Table се грижи за бизнес логиката, WeddingPlanner за наблюдението
- **Разширяемост:** Лесно добавяне на нови наблюдатели
- **Логиране на събития:** Проследяване на всички действия в системата

4. Command Pattern (Допълнителен шаблон)

Цел и предназначение:

Command Pattern енкапсулира заявка като обект, позволявайки параметризиране на клиенти с различни заявки, създаване на опашки от заявки и поддръжка на отменими операции.

Имплементация:

```
public class RelayCommand : ICommand
{
    private readonly Action _execute;
```

```
    private readonly Func<bool> _canExecute;
    public event EventHandler CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }

    public RelayCommand(Action execute, Func<bool> canExecute =
null)
    {
        _execute = execute ?? throw new
ArgumentNullException(nameof(execute));
        _canExecute = canExecute;
    }

    public bool CanExecute(object parameter) =>
_canExecute?.Invoke() ?? true;

    public void Execute(object parameter) => _execute();
}
```

Архитектурна структура на MVVM

Model-View-ViewModel (MVVM) е архитектурен шаблон, проектиран специално за приложения с графичен потребителски интерфейс. Той предлага ясно разделение на три основни компонента, като всеки от тях има специфична роля и отговорност. Целта на този разделителен подход е да се подобри поддръжката, тестваемостта и преизползваемостта на кода. MVVM следва принципа на "разделяй и владей" в софтуерната архитектура. Той трансформира традиционния подход "код зад интерфейса" в структурирана система, където бизнес логиката и потребителският интерфейс съществуват като независими, но комуникиращи си компоненти.

Model (Модел) - Фундаментът на данните

Model слоят представлява бизнес домейна и данните на приложението. Той съдържа:

- Бизнес обекти и тяхната структура
- Бизнес правила и валидации
- Логика за достъп до данни
- Алгоритми и изчисления

В контекста на проекта, Model включва всички обекти от предметната област: гости, семейства, маси и техните взаимоотношения. Този слой не знае нищо за потребителския интерфейс или начина на визуализация.

View (Изглед) - Визуалната репрезентация

View слоят е отговорен единствено за потребителското изживяване:

- Визуален дизайн и оформление
- Потребителски взаимодействия
- Анимации и преходи
- Отговорен дизайн

View не съдържа бизнес логика - той просто представя информацията и предава потребителските действия към ViewModel. В проекта, това е XAML файлът, който дефинира как изглежда приложението.

ViewModel (Модел-изглед) - Мостът между данни и интерфейс

ViewModel действа като посредник между Model и View:

- Подготвя данни за визуализация
- Обработва потребителски команди
- Поддържа състоянието на интерфейса
- Имплементира бизнес логика за презентация

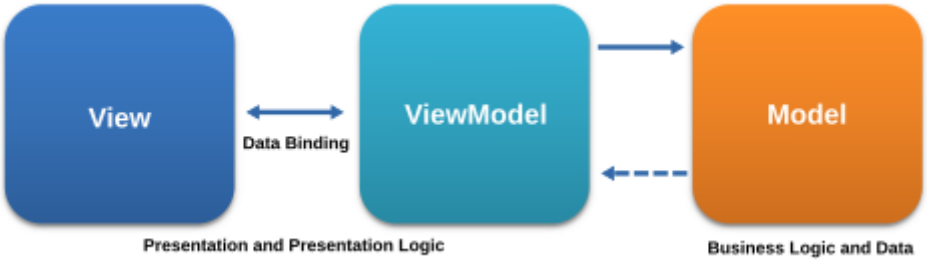
Това е "моделът на изгледа" - той предоставя данни и поведения, специфични за потребителския интерфейс, без да знае конкретните детайли за визуализацията.

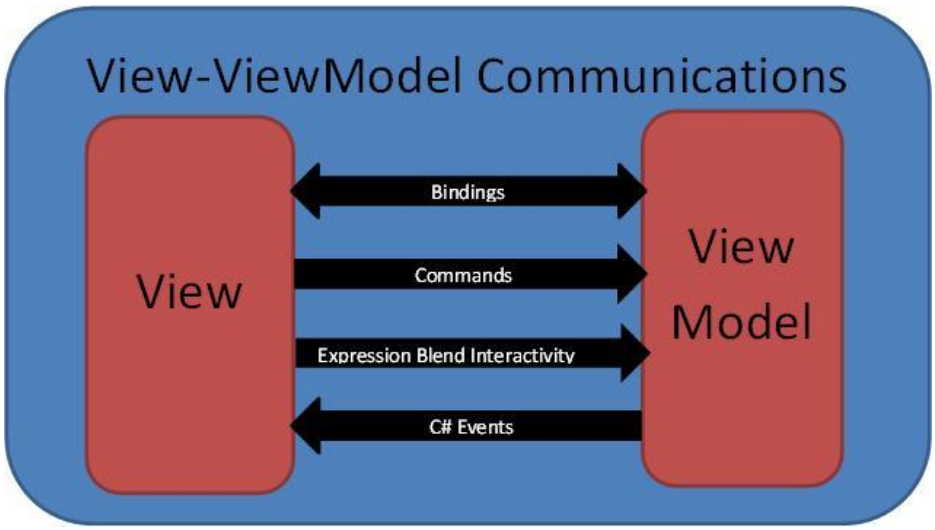
Data Binding

Data Binding е механизмът, който автоматично синхронизира данните между View и ViewModel. Той елиминира необходимостта от ръчно писане на код за актуализиране на интерфейса при промяна на данните и е ключов за софтуерни решения, които използват MVVM архитектура. В проекта се използват следните видове binding – **еднопосочен, двупосочен и command binding.**

Защо MVVM

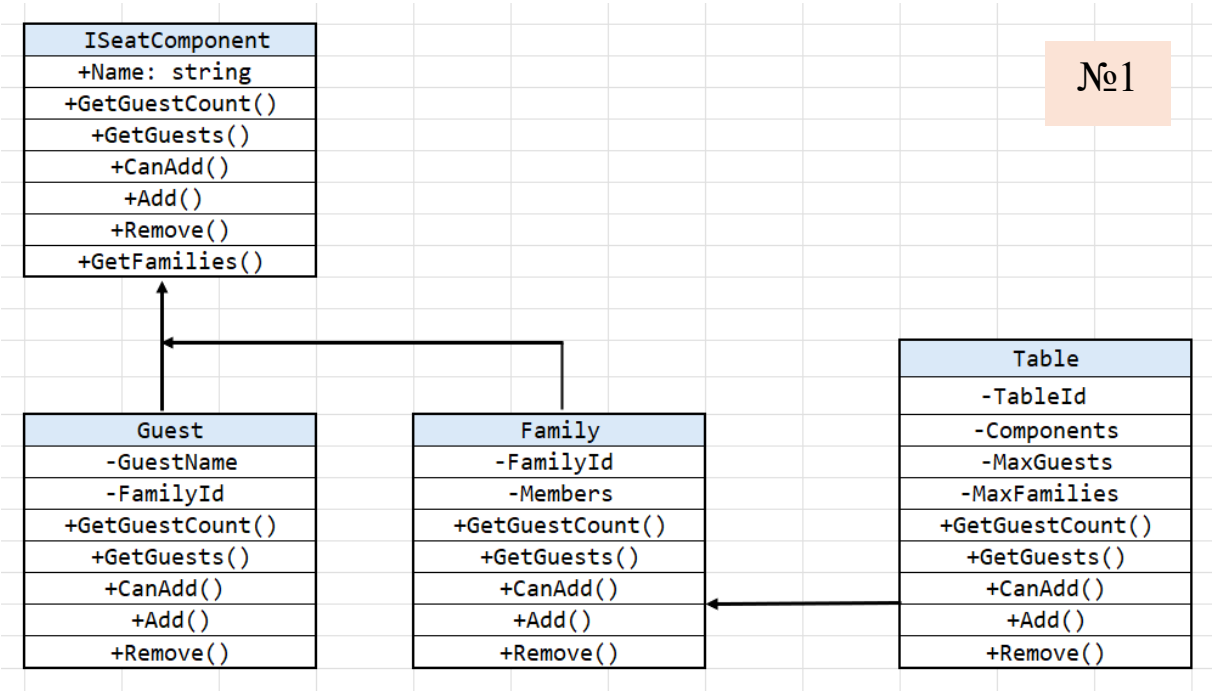
- **Лесна поддръжка и еволюция**
- **Преизползваемост на компоненти**
- **Ясна разделителна способност**
- **Вече сме го изучавали по дисциплината „Програмни среди“**

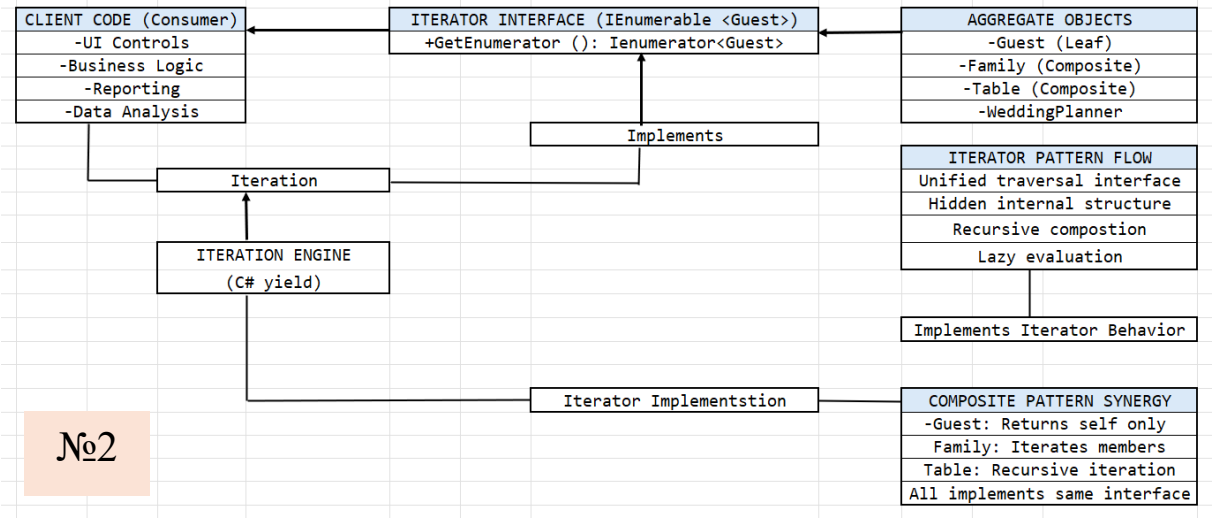


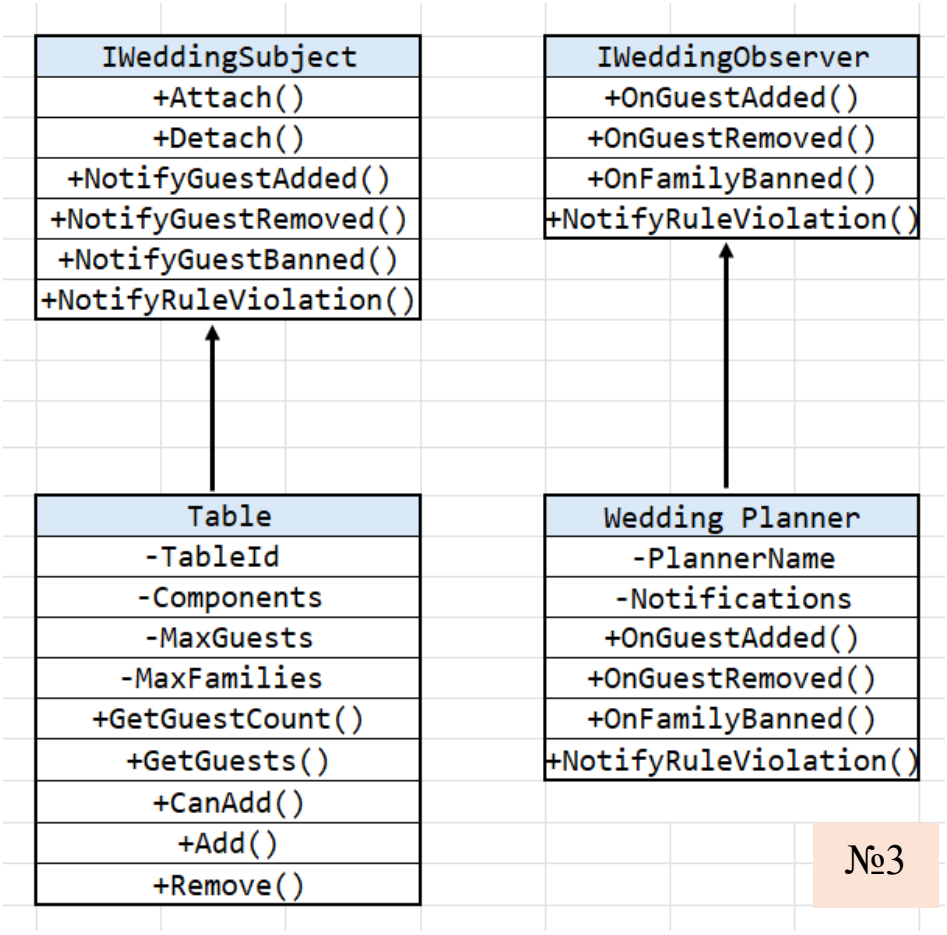


UML диаграми

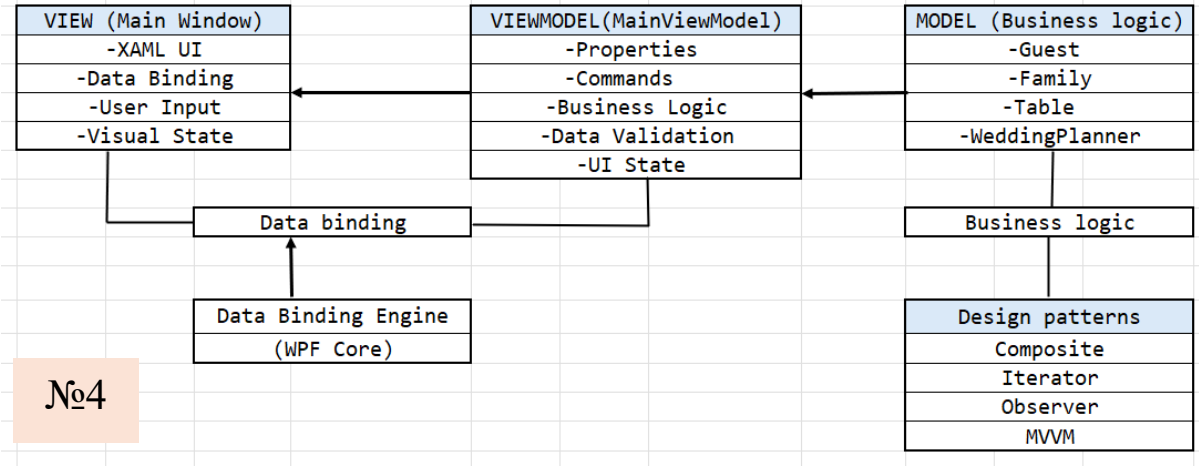
1.) Composite Pattern Structure
2.) Iterator Pattern Structure
3.) Observer Pattern Structure
4.) Class Diagram - MVVM Architecture
5.) Sequence Diagram - Assign Guest Process


ISeatComponent
+Name: string
+GetGuestCount()
+GetGuests()
+CanAdd()
+Add()
+Remove()
+GetFamilies()
№1
Guest
-GuestName
-FamilyId
+GetGuestCount()
+GetGuests()
+CanAdd()
+Add()
+Remove()
Family
-FamilyId
-Members
+GetGuestCount()
+GetGuests()
+CanAdd()
+Add()
+Remove()
Table
-TableId
-Components
-MaxGuests
-MaxFamilies
+GetGuestCount()
+GetGuests()
+CanAdd()
+Add()
+Remove()

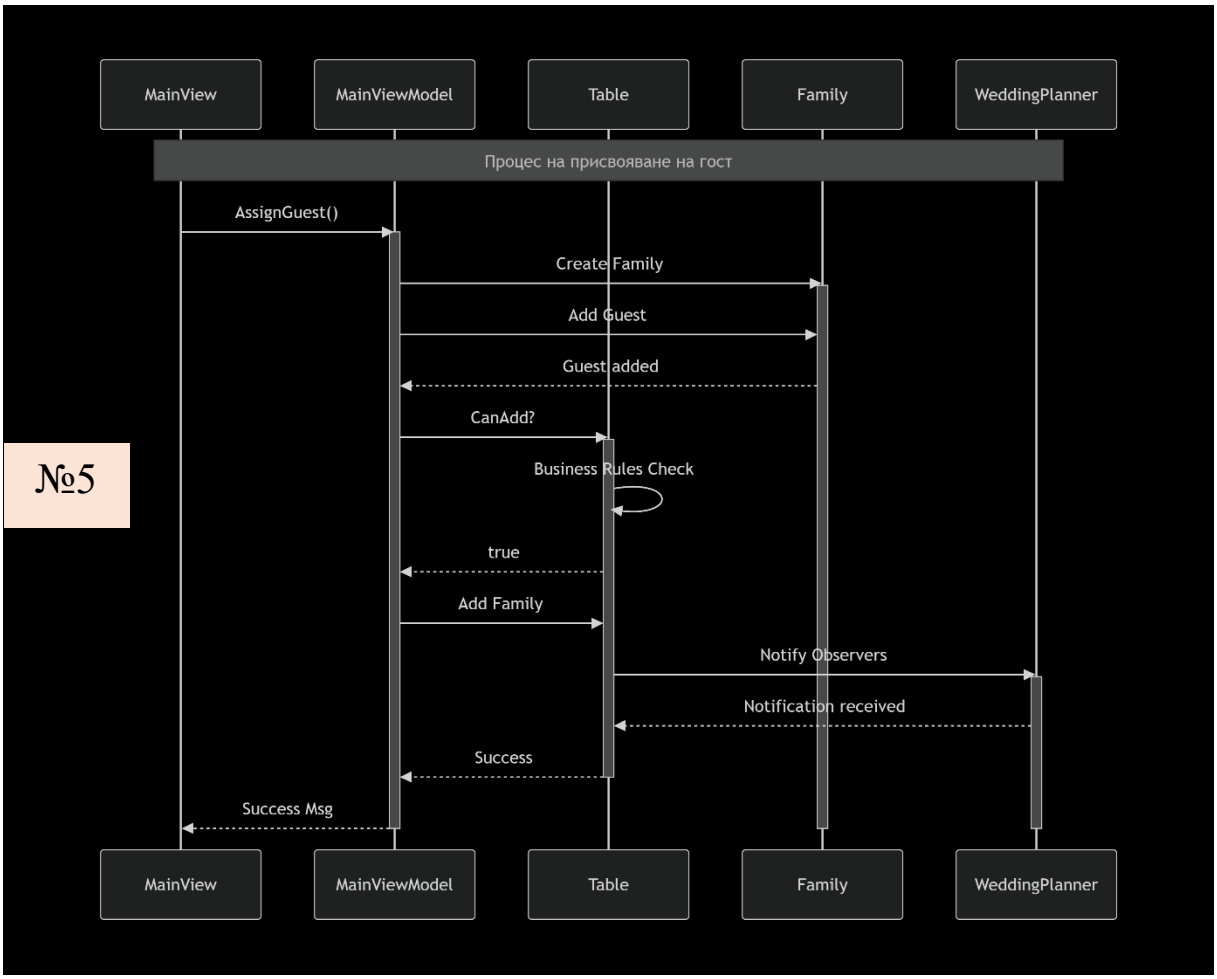

CLIENT CODE (Consumer)
-UI Controls
-Business Logic
-Reporting
-Data Analysis
ITERATOR INTERFACE (IEnumerable <Guest>)
+GetEnumerator (): Ienumerator<Guest>
AGGREGATE OBJECTS
-Guest (Leaf)
-Family (Composite)
-Table (Composite)
-WeddingPlanner
Implements
Iteration
ITERATION ENGINE
(C# yield)
ITERATOR PATTERN FLOW
Unified traversal interface
Hidden internal structure
Recursive compostion
Lazy evaluation
Implements Iterator Behavior
Iterator Implementstion
COMPOSITE PATTERN SYNERGY
-Guest: Returns self only
Family: Iterates members
Table: Recursive iteration
All implements same interface
№2


IWeddingSubject
+Attach()
+Detach()
+NotifyGuestAdded()
+NotifyGuestRemoved()
+NotifyGuestBanned()
+NotifyRuleViolation()
IWeddingObserver
+OnGuestAdded()
+OnGuestRemoved()
+OnFamilyBanned()
+NotifyRuleViolation()
Table
-TableId
-Components
-MaxGuests
-MaxFamilies
+GetGuestCount()
+GetGuests()
+CanAdd()
+Add()
+Remove()
Wedding Planner
-PlannerName
-Notifications
+OnGuestAdded()
+OnGuestRemoved()
+OnFamilyBanned()
+NotifyRuleViolation()

№3


VIEW (Main Window)
-XAML UI
-Data Binding
-User Input
-Visual State
VIEWMODEL(MainViewModel)
-Properties
-Commands
-Business Logic
-Data Validation
-UI State
MODEL (Business logic)
-Guest
-Family
-Table
-WeddingPlanner
Data binding
Business logic
Data Binding Engine
(WPF Core)
Design patterns
Composite
Iterator
Observer
MVVM

№4



Заключение

Wedding Seating Planner е перфектният софтуер за успешно създаване на сватбена организация за настаняване на гости от различни семейства. Интуитивен дизайн, мощни функции и интелигентна валидация гарантират, че всеки гост ще бъде на правилното място и няма да се получат недоразумения. Приложението успешно имплементира различни дизайн шаблони, които спомагат неговото съвършенство и възможност за лесно реализируемо бъдещо развитие.

Литература

- https://en.wikipedia.org/wiki/Software_design_pattern
- https://refactoring.guru/design-patterns/composite
- https://refactoring.guru/design-patterns/iterator
- https://refactoring.guru/design-patterns/observer
- https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm