ПОЛТАВСЬКИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТОРГІВЛІ
Навчально-науковий інститут денної освіти
Форма навчання денна
Кафедра комп'ютерних наук та інформаційних технологій

Допускається до захисту
Завідувач кафедри
_____ Олена ОЛЬХОВСЬКА
(підпис)

«___» _____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

на тему
ПРОГРАМНА РЕАЛІЗАЦІЯ ОСВІТНЬОГО СЕРЕДОВИЩА ДЛЯ ФОРМУВАННЯ МАТЕМАТИЧНИХ КОМПЕТЕНТНОСТЕЙ ЗА ДОПОМОГОЮ ІГРОВИХ ПІДХОДІВ

зі спеціальності 122 Комп'ютерні науки
освітня програма «Комп'ютерні науки»
ступеня магістра

Виконавець роботи Ветущенко Максим Леонідович
_____ «____»_____ 2025 р.
(підпис)

Науковий керівник к.пед.н., доцент, Кошова Оксана Петрівна
_____ «____»_____ 2025 р.
(підпис)

Рецензент

ПОЛТАВА 2025 р.

ЗАТВЕРДЖУЮ
Завідувач кафедри _____Олена ОЛЬХОВСЬКА
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ І КАЛЕНДАРНИЙ ГРАФІК
ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

на тему «Програмна реалізація освітнього середовища для формування математичних компетентностей за допомогою ігрових підходів**»**
зі спеціальності 122 Комп'ютерні науки
освітня програма «Комп'ютерні науки»
ступеня магістр
Прізвище, ім'я, по батькові Ветущенко Максим Леонідович
Затверджена наказом ректора № ____ _____ 2024 р..
Термін подання студентом роботи «___»_____ 2025 р.
Вихідні дані до кваліфікаційної роботи: статті та документації з теми розробки сервісів для створення навчальних ігрових застосунків.

Зміст пояснювальної записки (перелік питань, які потрібно розробити)
ВСТУП
РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ
РОЗДІЛ 2. ОГЛЯД СИСТЕМ АНАЛОГІЧНОГО ПРИЗНАЧЕННЯ
2.1. Огляд сучасних навчальних програм та ігор з математики і логіки.
2.2. Порівняльний аналіз програмних засобів для логіко-математичних ігор.
2.3. Аналіз інтерфейсів та дидактичних можливостей подібних систем.
2.4. Визначення недоліків існуючих рішень.
2.5. Обґрунтування необхідності створення програмної реалізації гри «Математико».
РОЗДІЛ 3. ЕОРЕТИЧНІ ТА АЛГОРИТМІЧНІ ОСНОВИ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ГРИ «МАТЕМАТИКО»
3.1. Математична модель гри «Математико».
3.2. Логіко-комбінаторні аспекти гри.
3.3. Алгоритми роботи комп'ютера-суперника.
3.4. Моделювання структури програмної системи та UML-діаграми.
РОЗДІЛ 4. ПРАКТИЧНА ЧАСТИНА
4.1. Обґрунтування вибору технологій реалізації.
4.2. Структура програмного продукту, реалізація ігрового поля та механізму ходів.
4.3. Реалізація алгоритмів комп'ютера-суперника.
4.4. Реалізація підрахунку очок та візуалізація результатів.
4.5. Інтерфейс користувача та особливості використання.
ВИСНОВКИ
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ
ДОДАТОК А

Перелік графічного матеріалу: 1 аркуш діаграм, 24 ілюстрації.

Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали, посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Постановка задачі	Кошова О.П.		
Інформаційний огляд	Кошова О.П.		
Теоретична частина	Кошова О.П.		
Практична частина	Кошова О.П.		

Календарний графік виконання кваліфікаційної роботи

Зміст роботи	Термін виконання	Фактичне виконання
Вступ		
Вивчення методичних рекомендацій та стандартів та звіт керівнику		
Постановка задачі		
Інформаційний огляд джерел бібліотек та інтернету		
Теоретична частина		
Практична частина		
Закінчення оформлення		
Доповідь студента на кафедрі		
Доробка (за необхідністю), рецензування		

Дата видачі завдання «__» _____ 2024 р.

Здобувач вищої освіти _____ Ветущенко Максим Леонідович
(підпис)

Науковий керівник _____ к.пед.н., доц. Кошова О.П.
(підпис) (науковий ступінь, вчене звання, ініціали та прізвище)

Результати захисту кваліфікаційної роботи

Кваліфікаційна робота оцінена на _____
(балів, оцінка за національною шкалою, оцінка за ECTS)

Протокол засідання ЕК № ____ від «____» _____ 2025 р.

Секретар ЕК_____ _____
(підпис) (ініціали та прізвище)

Затверджую
Зав. кафедрою _____
к.ф.-м.н. Олена ОЛЬХОВСЬКА
«____» _____ 2024 р.

Погоджено
Науковий керівник _____
к.пед.н., Оксана КОШОВА
«____» _____ 2024 р.

План

дипломного проекту з фаху
спеціальності 122 Комп'ютерні науки
освітня програма 122 Комп'ютерні науки
на тему «Програмна реалізація освітнього середовища для формування математичних компетентностей за допомогою ігрових підходів»
Прізвище, ім'я, по батькові Ветущенко Максим Леонідович

ВСТУП
РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ
РОЗДІЛ 2. ОГЛЯД СИСТЕМ АНАЛОГІЧНОГО ПРИЗНАЧЕННЯ
2.1. Огляд сучасних навчальних програм та ігор з математики і логіки.
2.2. Порівняльний аналіз програмних засобів для логіко-математичних ігор.
2.3. Аналіз інтерфейсів та дидактичних можливостей подібних систем.
2.4. Визначення недоліків існуючих рішень.
2.5. Обґрунтування необхідності створення програмної реалізації гри «Математико».
РОЗДІЛ 3. ЕОРЕТИЧНІ ТА АЛГОРИТМІЧНІ ОСНОВИ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ГРИ «МАТЕМАТИКО»
3.1. Математична модель гри «Математико».
3.2. Логіко-комбінаторні аспекти гри.
3.3. Алгоритми роботи комп'ютера-суперника.
3.4. Моделювання структури програмної системи та UML-діаграми.
РОЗДІЛ 4. ПРАКТИЧНА ЧАСТИНА
4.1. Обґрунтування вибору технологій реалізації.
4.2. Структура програмного продукту, реалізація ігрового поля та механізму ходів.
4.3. Реалізація алгоритмів комп'ютера-суперника.
4.4. Реалізація підрахунку очок та візуалізація результатів.
4.5. Інтерфейс користувача та особливості використання.
ВИСНОВКИ
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ
ДОДАТОК А

Здобувач вищої освіти _____ М.Л.Ветущенко

(підпис)

«____» _____ 2024 р.

РЕФЕРАТ

Записка: 106 сторінок, 24 рисунки, 1 додаток, 24 літературні джерела.

Метою даного дослідження є розробка програмної реалізації ігрового середовища «Математико» у вигляді десктопної комп’ютерної гри з комп’ютером-суперником, що включає математичну модель гри, алгоритми вибору ходів та підрахунку очок, архітектуру програмної системи й графічний інтерфейс користувача на основі технології WPF.

Об’єктом дослідження є процес побудови програмних систем, що реалізують логіко-математичні ігри з фіксованими правилами та алгоритмами комп’ютерного супротивника.

Предметом дослідження виступають методи та програмні засоби моделювання логіко-математичної гри «Математико», алгоритми вибору ходів і підрахунку очок, а також архітектурні підходи до побудови десктопних ігрових застосунків у середовищі C# / WPF.

У процесі розробки застосовувалися такі **методи**: модульне програмування, об’єктно-орієнтоване проєктування, UML-моделювання структури програмної системи, а також експериментальне тестування алгоритмів комп’ютера-суперника та підрахунку очок. Методологічну основу роботи становив об’єктно-орієнтований підхід і принципи структурного та компонентного проєктування, що забезпечило чітке розділення інтерфейсної та доменної логіки, а також можливість подальшого розширення функціональності. Інструментарій реалізації включав мову програмування C# та платформу .NET, технологію WPF для побудови графічного інтерфейсу, середовище розробки Visual Studio, а також засоби UML-моделювання (діаграми класів і послідовності) для фіксації архітектурних рішень.

Обґрунтовано доцільність створення окремої програмної реалізації гри «Математико» з відкритою структурою коду та прозорою математичною моделлю; формалізовано модель гри у вигляді заповнення матриці 5×5

значеннями з фіксованого мультисета та цільової функції, що враховує вклад рядків, стовпців і діагоналей; розроблено алгоритми підрахунку очок, які аналізують логіко-комбінаторні структури (послідовності, пари, трійки, четвірки, спеціальні комбінації); реалізовано евристичні алгоритми комп'ютера-суперника, що здійснюють вибір ходу на основі перебору можливих позицій та оцінювання конфігурацій; спроєктовано та побудовано UML-діаграми, які відображають структуру і динаміку роботи системи; реалізовано настільний застосунок із ігровим полем 5×5, механізмом ходів гравця та комп'ютера, модулем підрахунку очок і візуалізацією результатів; проведено тестування програмного продукту, яке підтвердило коректність роботи алгоритмів і відповідність реалізації поставленій меті дослідження.

ЗМІСТ

ВСТУП

Актуальність. Програмні системи з елементами гейміфікації посідають важливе місце серед засобів взаємодії користувача з інформацією. Розвиток інструментів для побудови десктопних та логіко-математичних ігор, графічних інтерфейсів користувача та інтелектуальних алгоритмів ухвалення рішень зумовлює потребу у створенні програмних продуктів, які поєднують зручність використання, надійну архітектуру та формалізовану ігрову логіку. Окремий інтерес становлять системи, що реалізують математично коректні моделі ігор із чітко визначеними правилами, можливістю автоматизованого підрахунку результатів та реалізацією алгоритмів комп'ютера-суперника. Такий підхід дозволяє розв'язувати низку прикладних задач інформатики: розробку ефективних структур даних, побудову алгоритмів пошуку та оцінювання станів, проєктування архітектури застосунку із використанням шаблонів проєктування та сучасних технологій розробки.

У цьому контексті актуальною є програмна реалізація логіко-математичної гри «Математико» як настільної комп'ютерної гри з ігровим полем фіксованого розміру, набором чисел та чітко визначеними правилами підрахунку очок за рядками, стовпцями та діагоналями. Її реалізація потребує побудови формальної математичної моделі, розробки алгоритмів вибору ходів комп'ютером-суперником, проєктування структури програмної системи, а також створення графічного інтерфейсу користувача із засобами візуалізації ігрових станів. Використання технологій C# та WPF, а також об'єктно-орієнтованого підходу дає змогу реалізувати масштабований застосунок із чітким розподілом відповідальності між компонентами та можливістю подальшого розширення функціональності.

Метою даного дослідження є розробка програмної реалізації ігрового середовища «Математико» у вигляді десктопної комп'ютерної гри з комп'ютером-суперником, що включає математичну модель гри, алгоритми

вибору ходів та підрахунку очок, архітектуру програмної системи й графічний інтерфейс користувача на основі технології WPF.

Об’єктом дослідження є процес побудови програмних систем, що реалізують логіко-математичні ігри з фіксованими правилами та алгоритмами комп’ютерного супротивника.

Предметом дослідження виступають методи та програмні засоби моделювання логіко-математичної гри «Математико», алгоритми вибору ходів і підрахунку очок, а також архітектурні підходи до побудови десктопних ігрових застосунків у середовищі C# / WPF.

Завдання дослідження полягають у наступному:

– проаналізувати логіко-математичні ігри та формалізувати постановку задачі для гри «Математико»;

– розробити концептуальну та математичну модель гри, що описує структуру ігрового поля, можливі стани та результати;

– спроєктувати архітектуру програмної системи, визначити основні компоненти та їх взаємодію;

– побудувати UML-діаграми для моделювання структури та поведінки програмного продукту;

– реалізувати програмний застосунок мовою C# з використанням технології WPF;

– реалізувати алгоритми гри та комп’ютерного супротивника, а також механізм підрахунку результатів;

– розробити графічний інтерфейс користувача та забезпечити його інтеграцію з логікою системи;

– здійснити тестування програмної системи та проаналізувати результати її роботи.

Запропонована програмна система базується на сучасних підходах до розробки десктопних ігор та використанні технологій .NET і WPF для побудови десктопних застосунків із графічним інтерфейсом. Реалізація гри

«Математико» у вигляді окремого застосунку з чітко виділеними модулями ігрового поля, логіки гри, комп’ютера-суперника та підрахунку результатів дозволяє дослідити питання проєктування архітектури, організації взаємодії компонентів і реалізації алгоритмів аналізу стану гри.

У процесі розробки застосовувалися методи модульного програмування, об’єктно-орієнтованого проєктування, моделювання структури та поведінки програмних систем за допомогою UML-діаграм, а також методи функціонального та модульного тестування. Інструментарій реалізації включав середовище розробки Visual Studio, мову програмування C#, платформу .NET, технологію WPF для побудови інтерфейсу користувача та XAML для опису візуальних компонентів.

Пояснювальна записка включає чотири основних розділи: постановку задачі, огляд систем аналогічного призначення, теоретичні та алгоритмічні основи програмної реалізації гри «Математико», а також практичну частину, присвячену опису структури, реалізації та тестування програмного продукту.

РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ

Основою для розробки програмної системи, що реалізує логіко-математичну гру «Математико» у вигляді настільного програмного застосунку є правила однойменної гри, у якій гравець та його супротивник послідовно розміщують числа на комірках ігрового поля з метою отримання максимальної кількості очок. Для комп’ютерної реалізації така гра розглядається як дискретна динамічна система з фіксованою структурою ігрового поля, заданим множинним набором чисел і визначеними правилами формування результату.

У дипломній роботі буде розглянуто варіант гри «Математико» з прямокутним ігровим полем розміром 5×5, кожна комірка якого може містити одне ціле число або залишатися порожньою. Набір чисел, що використовуються у грі, складається з елементів від 1 до 13, кожне з яких присутнє у фіксованій кількості копій. У процесі гри гравець і комп’ютер по черзі обирають поточне число та розміщують його у вільній комірці свого ігрового поля. Після заповнення картки здійснюється підрахунок очок, що базується на аналізі значень у рядках, стовпцях і діагоналях. Таким чином, початкові дані для програмної системи визначаються конфігурацією ігрового поля та множиною доступних чисел, а кінцевими результатами є сформований стан поля та обчислене підсумкове число очок для кожного гравця.

Задача програмної реалізації полягає в тому, щоб забезпечити коректне відображення ігрового поля, підтримку послідовності ходів, контроль допустимості дій користувача та автоматичний підрахунок очок згідно з правилами гри. Для цього ігрове поле доцільно представити у вигляді матриці розміру 5×5, елементами якої є значення, що відповідають числам, розміщеним гравцем або комп’ютером, чи позначення порожніх комірок. Простір станів гри задається всіма можливими конфігураціями цієї матриці з урахуванням множини вже використаних чисел. Окрему увагу слід приділити забезпеченню

цілісності даних: програма не повинна допускати розміщення числа у вже зайняту комірку, дублювання ходу або порушення послідовності етапів гри.

Однією з центральних підзадач є розробка алгоритмів вибору ходу комп’ютером-суперником. Для цього необхідно формально описати процедури аналізу поточного стану ігрового поля: оцінювання можливих розміщень числа з точки зору майбутнього підрахунку очок, врахування значень у діагоналях, рядках і стовпцях, а також визначення пріоритетних ходів. У програмній реалізації ці процедури мають бути інкапсульовані в окремому модулі логіки комп’ютера, який отримує на вхід поточний стан картки та значення числа, що необхідно розмістити, і повертає координати комірки, обраної для ходу. Така постановка задачі передбачає побудову алгоритмів, що поєднують перебір можливих варіантів із евристичними міркуваннями, а також можливість використання резервної стратегії випадкового ходу в ситуаціях, коли кілька варіантів мають однакову оцінку.

На рівні програмної системи постає задача проєктування архітектури застосунку з чітким розподілом відповідальності між компонентами. Необхідно виділити модуль відображення ігрового поля у вигляді графічного інтерфейсу користувача, модуль логіки гри, що відповідає за послідовність ходів, перевірку коректності дій та зміну стану поля, модуль підрахунку очок, який реалізує правила аналізу рядків, стовпців та діагоналей, а також модуль комп’ютера-суперника. Додатково потрібно передбачити компоненти для ініціалізації нової гри, зберігання поточного стану, відображення проміжних і фінальних результатів. Важливо, щоб взаємодія між модулями була формально описана та відображена за допомогою діаграм класів і діаграм послідовності, що дозволяє забезпечити прозорість та розширюваність архітектури.

У якості основної платформи обрана технологія WPF на базі .NET та мови програмування C#, що зумовлює використання механізмів подієво-орієнтованого програмування, розмітки інтерфейсу засобами XAML та зв’язування візуальних компонентів із логікою застосунку. Програмна система

повинна забезпечувати стабільну роботу в середовищі настільного застосунку, коректну обробку дій користувача, візуальне відображення стану гри та своєчасне оновлення інтерфейсу у відповідь на зміни внутрішніх даних. До не функціональних вимог належать надійність, передбачуваність результатів для однакових вхідних даних, зручність використання, а також можливість модифікації правил гри або параметрів ігрового поля без суттєвої перебудови всієї системи [1, 6-8, 11, 13].

Отже, у загальному вигляді постановку задачі полягає у необхідності розробки програмного настільного застосунку, що реалізує логіко-математичну гру «Математико» з фіксованими правилами, забезпечує графічне відображення ігрового поля, підтримує взаємодію користувача з системою та містить алгоритмічно обґрунтовану реалізацію комп’ютера-суперника й механізму підрахунку очок. У межах даної роботи будуть описані формальна модель гри, алгоритмічні рішення, архітектура програмного продукту та результати тестування, які підтверджують коректність і працездатність розробленої системи.

РОЗДІЛ 2. ОГЛЯД СИСТЕМ АНАЛОГІЧНОГО ПРИЗНАЧЕННЯ

2.1. Огляд сучасних навчальних програм та ігор з математики і логіки.

У сучасних умовах розвитку цифрових технологій навчальні програми та ігри з математики і логіки посідають важливе місце серед програмного забезпечення освітнього призначення. Нами було проаналізовано низку програмних продуктів, які реалізують різні підходи до формування математичних умінь: від тренажерів базових обчислювальних навичок до комплексних ігрових середовищ із елементами сюжетної лінії, змагання та адаптивного налаштування складності. Спільною рисою більшості таких систем є використання інтерактивних завдань, що подаються у формі коротких вправ, головоломок або ігрових ситуацій, які потребують від користувача виконання дій, пов’язаних із числовими операціями, логічними висновками чи аналізом структур.

Серед розповсюджених рішень слід відзначити платформу Prodigy Math Game, яка реалізує адаптивне тренування математичних навичок у форматі рольової гри з квестами, боями та накопиченням очок і винагород. Учні виконують математичні завдання, щоб здобувати перемоги в ігровому середовищі, а сама система позиціонується як безкоштовна для шкіл та батьків, із можливістю розширеного функціоналу за підпискою [15].

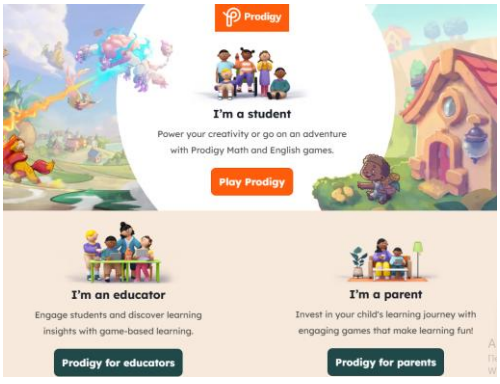

Рисунок 2.1. Платформа Prodigy Math Game

Подібний підхід дозволяє поєднати традиційні вправи з механікою пригодницької гри, що підвищує залученість користувачів і сприяє регулярному виконанню завдань.

Окрему групу становлять спеціалізовані математичні ігри серії DragonBox, які орієнтовані на візуалізацію алгебраїчних понять, геометрії та роботи з числами. Зокрема, ігри DragonBox Algebra 5+ та DragonBox Algebra 12+ поступово вводять поняття змінних і рівнянь у формі карткових головоломок, де користувач, виконуючи ігрові дії, фактично опановує операції, необхідні для розв’язування лінійних рівнянь [16-18].

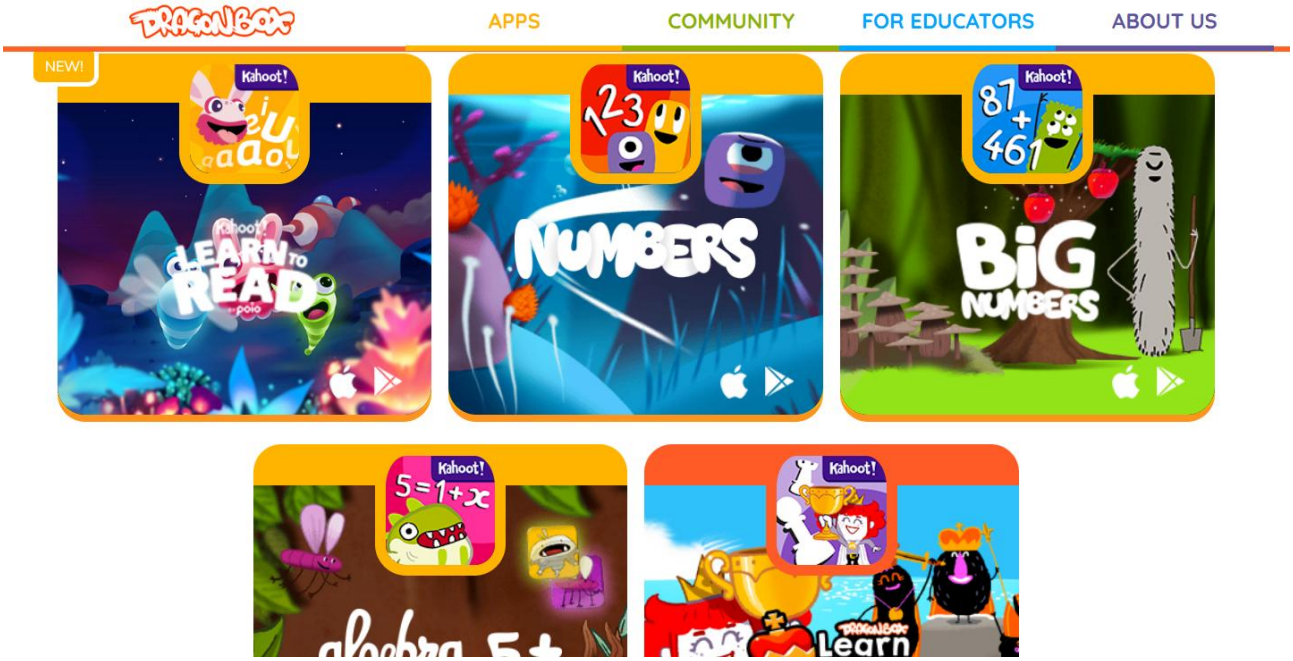

Рисунок 2.2. Математичні ігри серії DragonBox

Такі продукти демонструють можливість прихованого, але методично продуманого впровадження математичних понять у механіку гри. Також до цієї групи належать ігри Kahoot! Numbers та Kahoot! Algebra, створені на базі DragonBox, які надають дітям інтерактивне середовище для опанування базових числових операцій та елементів алгебри у форматі мобільних застосунків [18].

Риунок 2.3. ігри Kahoot! Numbers та Kahoot! Algebra

Суттєву нішу займають платформи, які поєднують масив навчальних матеріалів з елементами гейміфікації та адаптивного навчання. Прикладом є система Matific, що пропонує тисячі ігрових завдань і вправ з математики для молодших класів, організованих у вигляді «островів» та пригодницьких сценаріїв; система використовує адаптивні алгоритми, які підлаштовують складність до індивідуального темпу учня [19].

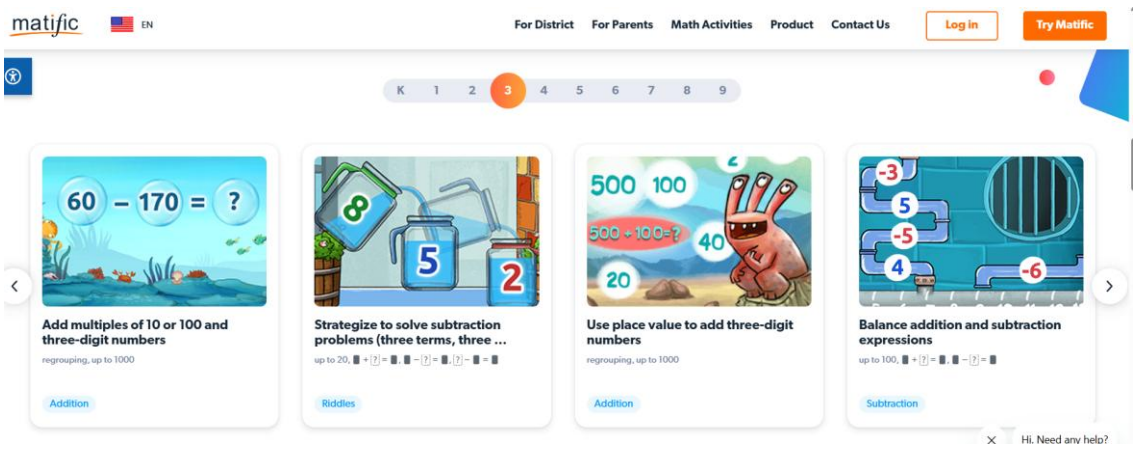

Рисунок 2.4. Система Matific

Ігровий інтерфейс доповнюється системою звітів для вчителів і батьків, що дає змогу поєднати практичну спрямованість завдань із контролем засвоєння матеріалу.

Поряд із спеціалізованими ігровими платформами нами було розглянуто

й універсальні освітні ресурси, такі як Khan Academy, яка надає велику бібліотеку відеолекцій та інтерактивних вправ з математики від початкових до університетських курсів, а також систему накопичення балів і значків за виконані завдання [20].

Хоча основний акцент тут робиться не на ігровому процесі, а на структурованому навчанні, присутні елементи гейміфікації та візуалізації прогресу. Іншим прикладом є платформа Brilliant, яка пропонує курси з математики та логіки з акцентом на розв'язуванні задач і логічних головоломок; користувачеві пропонуються послідовні інтерактивні завдання, що формують уміння робити дедуктивні висновки та застосовувати математичні методи в нових ситуаціях [21].

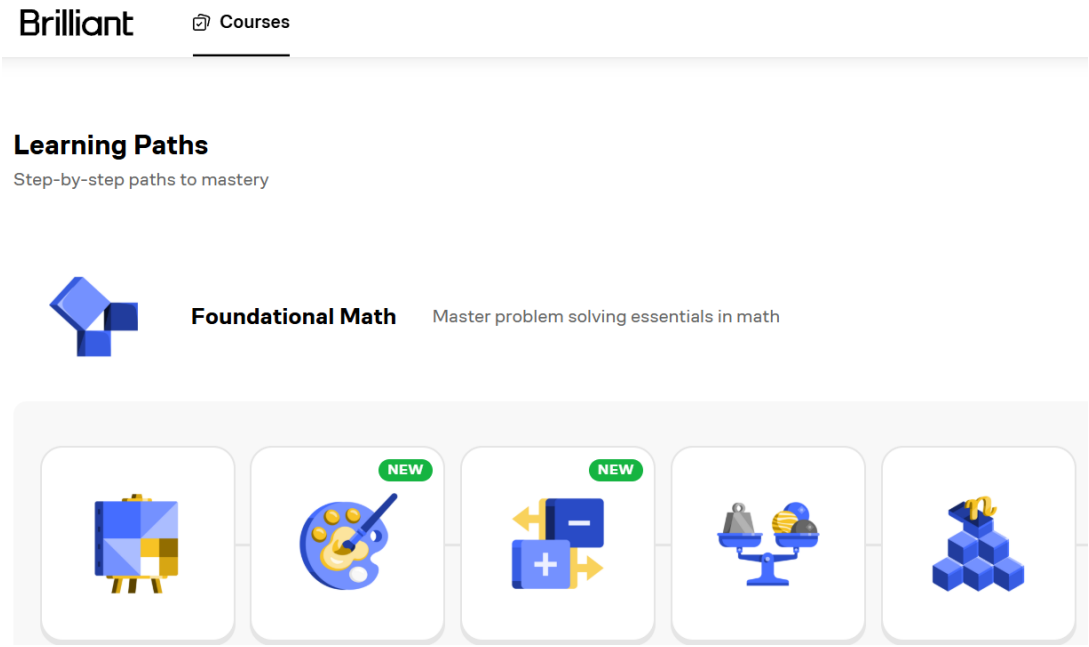

Рисунок 2.5 Платформа Brilliant

Окремо нами було відзначено численні мобільні ігри-головоломки, що не завжди позиціонуються як навчальні, але мають виразний логіко-математичний компонент: цифрові версії судоку, головоломки на заповнення сіток числами за певними правилами, ігри на пошук закономірностей і оптимізацію розміщення об'єктів. Хоча в них здебільшого відсутня формальна освітня складова у вигляді теоретичних пояснень чи структурованих курсів, механіка ігор часто

передбачає оперування числовими відношеннями, просторовими структурами та логічними умовами, що опосередковано сприяє розвитку математичного мислення.

Проведений огляд дає змогу зробити висновок, що сучасні навчальні програми та ігри з математики і логіки реалізують широкий спектр підходів: від повністю гейміфікованих ролевих світів, де математика вбудована в квестову механіку (Prodigy, Matific), до спеціалізованих середовищ для опанування окремих математичних концепцій (DragonBox, Kahoot! Algebra) та універсальних навчальних платформ із елементами гейміфікації (Khan Academy, Brilliant). Разом із тим, у більшості з проаналізованих рішень логіка підрахунку результатів та алгоритми «супротивника» або спрощені, або залишаються прихованими від користувача, що обмежує можливості їх використання як об’єкта для вивчення алгоритмічних аспектів гри. У цьому контексті програмна реалізація гри «Математико», яку ми розробляємо, орієнтована на чітко формалізовану математичну модель, явну реалізацію алгоритмів аналізу стану ігрового поля та прозору систему підрахунку очок, що дозволяє розглядати її не лише як ігровий продукт, а й як об’єкт дослідження в галузі комп’ютерних наук.

Окремо нами було проаналізовано низку україномовних ресурсів, що пропонують інтерактивні завдання та ігри з математики й логіки. Зокрема, освітня платформа Learning.ua надає широкий спектр інтерактивних вправ, логічних ігор та головоломок для дошкільнят і школярів, серед яких окремо виділено завдання з логіки та математичні ігри для різних вікових категорій [22].

Рисунок 2.6. Освітня платформа Learning.ua

Проєкт «Алаба» (alaba.op.ua) орієнтований на освітні інтерактивні ігри, конкурси й олімпіади для учнів, містить україномовні ігри з математики та логіки, зокрема ігри-пазли та тренажери, розроблені відповідно до освітніх стандартів МОН України [23].

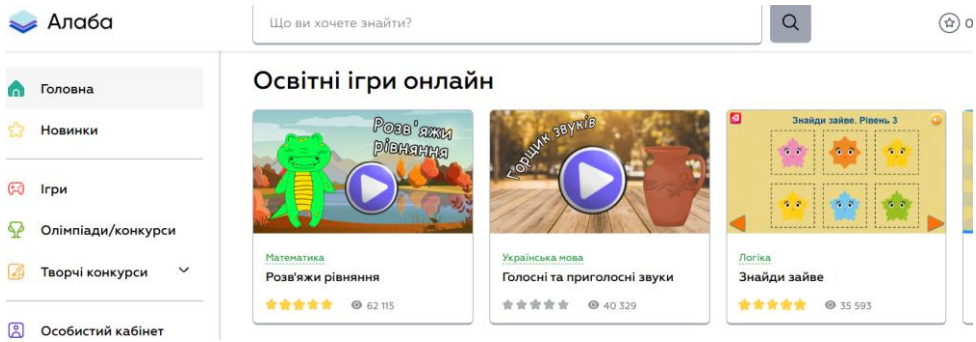

Рисунок 2.7. Проєкт «Алаба»

Окремий інтерес становить сайт Formula.co.ua, де в розділі «Логічні ігри он-лайн» представлено низку математичних ігор, таких як «Чотири арифметичні дії» чи «П’ятнашки», які спрямовані на тренування обчислювальних навичок і логічного мислення у форматі міні-ігор [24].

Хоча згадані ресурси пропонують різноманітні завдання й міні-ігри, їх основний акцент робиться на тренажерній і конкурсній складовій, тоді як формалізовані настільні логіко-математичні ігри з чітко описаною математичною моделлю та алгоритмічною реалізацією комп’ютерного супротивника представлені значно меншою мірою, що додатково обґрунтовує

доцільність розробки програмної реалізації гри «Математико».

2.2. Порівняльний аналіз програмних засобів для логіко-математичних ігор

У процесі дослідження нами було проаналізовано низку сучасних програмних засобів, що реалізують логіко-математичні ігри, з акцентом на використовувані технології, архітектурні підходи та особливості програмної реалізації. Загалом такі системи можна умовно поділити на три основні групи: веб-застосунки, мобільні додатки та настільні (desktop) програми. Кожна з цих груп характеризується власним типовым стеком технологій і відповідними компромісами між продуктивністю, зручністю розгортання та можливостями інтеграції з іншими сервісами.

До першої групи належать веб-орієнтовані навчальні ігри та платформи, які працюють безпосередньо в браузері. Типовою є реалізація клієнтської частини на основі HTML5, CSS та JavaScript із використанням додаткових бібліотек і фреймворків, зокрема React, Vue або Angular, а також технологій Canvas чи WebGL для побудови інтерактивної графіки. Такі рішення забезпечують кросплатформеність, оскільки одна і та сама гра може запускатися на різних операційних системах за умови наявності сучасного браузера. На стороні сервера зазвичай застосовуються Node.js, Python або Java разом із веб-фреймворками для організації бізнес-логіки, збереження прогресу користувачів, обробки аналітики та інтеграції з базами даних. Архітектура подібних систем орієнтована на багатокористувацький доступ, підтримку облікових записів, рейтингових таблиць і масштабування під велике навантаження [14].

Друга велика група програмних продуктів представлена мобільними

додатками для платформ Android та iOS. Для їх розробки широко використовуються нативні засоби (Java або Kotlin для Android, Swift або Objective-C для iOS), а також кросплатформені фреймворки, такі як Unity, Flutter чи React Native. У випадку логіко-математичних ігор на перший план виходить підтримка плавної анімації, жестів, сенсорного введення, що визначає вибір рушіїв із розвиненими засобами роботи з графікою та подіями. Такі додатки часто побудовані за клієнт–серверною схемою: логіка гри частково реалізується на клієнті, тоді як збереження досягнень, рейтинги та аналітика винесені на серверну сторону. Мобільний формат стимулює створення коротких за часом сесій, що впливає на структуру ігрових рівнів і механіку нарахування балів [14].

Третю групу складають настільні (desktop) логіко-математичні ігри, що розгортаються як окремі застосунки під керуванням операційних систем сімейства Windows, Linux або macOS. Для їх реалізації використовуються різні технологічні стеки: Java (Swing, JavaFX), C++ з бібліотеками Qt або SFML, C# з WinForms або WPF, а також універсальні ігрові рушії, такі як Unity чи Godot, здатні генерувати окремі збірки для ПК. Настільні застосунки, на відміну від браузерних і хмарних рішень, частіше орієнтовані на локальне збереження стану гри та автономну роботу без постійного підключення до мережі, що спрощує архітектуру, але водночас обмежує можливості централізованого збору статистики й організації онлайн-взаємодії користувачів [14].

Порівняльний аналіз показав, що у більшості комерційних і навчальних логіко-математичних ігор значна частина зусиль розробників спрямована на створення масштабованої інфраструктури та сервісів: систем реєстрації й авторизації, платіжних модулів, механізмів аналітики та гейміфікації (досягнення, рейтинги, внутрішня валюта, мікротранзакції). Водночас математична модель гри та алгоритми супротивника здебільшого залишаються прихованими від кінцевого користувача, а інтерфейс, як правило, не дозволяє дослідникові безпосередньо аналізувати внутрішні структури даних або логіку

прийняття рішень. Для цілей навчальних курсів із програмування та комп’ютних наук це створює певні обмеження: готові системи є цікавими як приклад комплексних хмарних рішень, але малопридатні як прозорі моделі для вивчення алгоритмів і структури коду.

На цьому тлі розроблений нами програмний застосунок «Математико» має інший акцент. Він створений як настільна програма для середовища Windows із використанням мови програмування C# та технології WPF. Такий вибір зумовив низку особливостей. По-перше, графічний інтерфейс реалізовано засобами XAML та користувацьких елементів керування, що дозволило чітко відокремити візуальне представлення ігрового поля від логіки його роботи. Для цього було створено окремі компоненти, зокрема користувацький елемент для відображення ігрових карток та сервісні класи, що відповідають за логіку комп’ютера-суперника і підрахунок очок. По-друге, архітектура застосунку побудована як локальна, без використання віддалених серверів або хмарних сервісів, що спрощує розгортання та робить програму самодостатньою в контексті навчальних аудиторій чи індивідуального використання [1, 4-8, 9-13].

З технічної точки зору розроблений застосунок тяжіє до моделі настільної логіко-математичної гри з чітко формалізованими правилами, явною структурою класів і прозорою реалізацією алгоритмів. На відміну від багатьох веб-орієнтованих платформ, де клієнтська частина є лише графічною оболонкою для серверної логіки, в даному випадку всі ключові обчислення (аналіз рядків, стовпців і діагоналей, вибір ходу комп’ютера, підрахунок очок) виконуються локально й представлені у вигляді окремих класів і методів у кодовій базі. Це створює можливості для детального вивчення, модифікації та розширення алгоритмів без необхідності зміни інфраструктурної складової.

Порівнюючи стек технологій і архітектурні підходи, можна зазначити, що, на відміну від багатошарових клієнт–серверних систем із використанням веб-фреймворків, мікросервісів і мобільних SDK, реалізація гри «Математико» у середовищі C#/WPF є відносно простою за структурою, але більш прозорою з

точки зору архітектури й коду. У результаті розроблений застосунок може розглядатися як приклад програмного засобу, що поєднує достатньо сучасний технологічний стек із навчальною цінністю з точки зору аналізу архітектури та алгоритмів логіко-математичної гри.

2.3. Аналіз інтерфейсів та дидактичних можливостей подібних систем

Аналіз інтерфейсних рішень та дидактичних можливостей сучасних логіко-математичних ігор дає змогу виявити типові підходи до організації взаємодії користувача з програмним продуктом і вбудовування навчального змісту в ігрову діяльність. Нами було розглянуто низку популярних систем, зокрема Prodigy Math Game, Matific, DragonBox, а також україномовні ресурси на кшталт Learning.ua, що дозволило порівняти різні моделі побудови користувацьких інтерфейсів та механізмів навчального впливу [15-24].

У веб- та онлайнових платформах, таких як Prodigy Math, інтерфейс побудовано у форматі віртуального ігрового світу з 3D або псевдо-3D середовищем, де навчальні завдання інтегруються в сюжетні квести, бої та дослідження локацій. Користувач керує персонажем, який пересувається простором, взаємодіє з об’єктами та іншими персонажами, а математичні вправи подаються як умова для виконання дії чи переходу на новий рівень. Реалізовано розвинену систему візуального зворотного зв’язку: прогрес відображається через шкали досвіду, віртуальні нагороди, нове екіпірування тощо, що підсилює мотиваційну складову й сприяє тривалому утриманню уваги користувача. Дидактичні можливості таких систем базуються на адаптивній подачі завдань: рівень складності й тип вправ добираються автоматично на основі успішності гравця. Водночас математичний зміст часто подається у вигляді окремих прикладів, «вбудованих» у загальну механіку рольової гри, що не завжди дозволяє фокусувати увагу на структурі математичної моделі.

Платформи типу Matific використовують дещо інший підхід: інтерфейс організовано у вигляді набору міні-ігор та інтерактивних сцен, де користувач виконує конкретні дії (перетягування об’єктів, позначення величин, розміщення елементів на числових променях тощо). Кожна така сцена є самодостатнім завданням із чітко визначеною метою, а система надає негайний зворотний зв’язок про правильність дій, відображаючи помилки й пропонуючи підказки. Навчальний контент структуровано у вигляді послідовностей завдань, які користувач проходить у власному темпі, а адаптивні алгоритми забезпечують добір відповідного рівня складності. Дидактичні можливості таких інтерфейсів полягають у поєднанні візуалізації математичних об’єктів із поетапним ускладненням завдань; однак логіка побудови ігор і спосіб нарахування балів у більшості випадків залишається непрозорим для користувача з точки зору внутрішніх структур даних та алгоритмів.

У спеціалізованих іграх серії DragonBox інтерфейс орієнтований на максимально інтуїтивну візуалізацію алгебраїчних конструкцій. Користувач взаємодіє з картками-символами, маніпулюючи ними згідно з правилами, які поступово розкриваються в процесі гри; таким чином, формальні операції над рівняннями подаються у вигляді серії візуальних. Інтерфейс мінімізує текстові пояснення, натомість покладається на експериментальне відкриття правил, що сприяє розвитку інтуїтивного розуміння, але ускладнює експлікацію математичної моделі для подальшого програмного аналізу.

Україномовні ресурси на кшталт Learning.ua використовують більш традиційні веб-інтерфейси: екран поділено на область завдання, робочу зону та панель результатів; навчальні завдання подаються у вигляді тестових запитань, прикладів або простих ігрових сцен із накопиченням балів, відсотків виконання й віртуальних нагород. Дидактичні можливості забезпечуються через систематичне покриття шкільної програми, різнорівневі завдання й наявність звітів для вчителів і батьків. Однак і тут домінують короткі, мало пов’язані між собою вправи-тренажери, а не комплексні логіко-математичні ігри з

послідовністю станів і стратегічною взаємодією гравця з алгоритмічним супротивником.

З погляду інтерфейсних рішень можна виділити кілька спільних рис проаналізованих систем. По-перше, майже в усіх з них значну увагу приділено візуальній привабливості: використовується яскрава кольорова гама, анімовані персонажі, іконки, нагороди, що підсилюють мотиваційний аспект, але не завжди сприяють концентрації саме на логіко-математичній структурі завдання. По-друге, система зворотного зв’язку зазвичай подається у вигляді сповіщень про правильність/неправильність відповіді, відсотків виконання, віртуальних медалей та сертифікатів, тоді як пояснення помилок, деталізований розбір рішень і можливість проаналізувати альтернативні ходи супротивника присутні обмежено. По-третє, у більшості продуктів інтерфейс орієнтований на одномоментну взаємодію з окремим завданням або коротким рівнем гри; складні багатокрокові стратегії трапляються рідше, а внутрішня модель гри рідко виступає об’єктом для явного осмислення користувачем.

На цьому фоні інтерфейс розробленого нами застосунку «Математико» має іншу спрямованість. Основним елементом є ігрове поле розміром 5×5, яке представляється у вигляді таблиці клітинок з чітко фіксованими позиціями, у кожну з яких може бути розміщене конкретне число. Інтерфейс побудовано засобами WPF із використанням користувацького елемента, що відповідає за відображення окремих «карток» гри; натомість логіка розміщення чисел, перевірка допустимості ходу, підрахунок очок і робота комп’ютера-суперника реалізуються у сервісних класах. Такий підхід забезпечує візуальну простоту й структурованість: користувач у будь-який момент бачить повну конфігурацію ігрового поля, а також поточний рахунок, що напряму відображає результати попередніх дій. На відміну від багатьох веб- та мобільних ігор, де математичний зміст “розчинено” в сюжеті або міні-іграх, у нашому застосунку інтерфейс безпосередньо відображає математичну модель: матрицю значень, які впливають на підсумковий результат через аналіз рядків, стовпців і

діагоналей.

Дидактичні можливості такого інтерфейсу полягають не стільки в розгалуженій системі нагород чи сюжетних механіках, скільки у прозорості зв’язку між діями користувача й підсумковим результатом. Користувач має змогу спостерігати, як зміна позиції конкретного числа на полі впливає на суму очок, і реконструювати логіку підрахунку. Оскільки алгоритми роботи комп’ютерного супротивника реалізовано у вигляді окремих класів, дослідник або розробник може аналізувати й модифікувати їх, експериментуючи з різними стратегіями, не змінюючи при цьому інтерфейсну складову. Таким чином, розроблений застосунок, будучи простішим за візуальним оформленням порівняно з комерційними платформами на кшталт Prodigy чи Matific, водночас має перевагу з точки зору навчального використання в галузі комп’ютерних наук: його інтерфейс і внутрішня логіка побудовані таким чином, щоб чітко віддзеркалювати математичну модель і алгоритми гри, що полегшує їх аналіз, дослідження та подальшу модернізацію.

2.4. Визначення недоліків існуючих рішень

На основі проведеного нами огляду та аналізу сучасних навчальних програм і логіко-математичних ігор можна виокремити низку характерних недоліків, які мають як суто програмно-інженерну, так і методологічну природу. Ці недоліки є ключовими аргументами на користь розроблення окремого програмного засобу, що реалізує логіко-математичну гру «Математико» із чітко формалізованою внутрішньою моделлю та прозорими алгоритмами.

По-перше, для більшості розглянутих нами систем є типовою орієнтація на зовнішню привабливість інтерфейсу, розгалужені механізми гейміфікації (нагороди, рейтинги, віртуальні бонуси), інтеграцію з веб-сервісами та

мобільними платформами при відносно обмеженій відкритості внутрішньої математичної моделі. Правила підрахунку очок, умови формування результатів і логіка прийняття рішень алгоритмічним супротивником, як правило, є прихованими для користувача й не документуються в явному вигляді. Це створює суттєві труднощі для дослідників і розробників, які прагнуть використовувати ці системи як об’єкти вивчення алгоритмів і архітектурних рішень: поведінка системи сприймається як «чорна скринька», де результат є відомим, але механізм його отримання залишається непрозорим.

По-друге, значна частина навчальних та ігрових платформ реалізована як веб- або хмарні сервіси з клієнт–серверною архітектурою. Такий підхід виправданий з погляду масштабування, багатокористувацького доступу й інтеграції з аналітичними інструментами, однак він обумовлює низку обмежень. Джерельний код ядра системи та алгоритмів супротивника недоступний для локального вивчення; навіть за наявності відкритих API розробник має справу лише з результатами віддалених викликів, не маючи змоги детально дослідити внутрішні структури даних і реалізацію алгоритмів. Для освітніх програм у галузі комп’ютерних наук це означає, що такі продукти можна показувати як приклад роботи складної інфраструктури, але не використовувати як повноцінні тренажери для аналізу програмної реалізації логіко-математичних ігор.

По-третє, у багатьох існуючих рішеннях логіко-математичні завдання подаються у вигляді набору окремих міні-ігор або тренажерів, слабко пов’язаних між собою з точки зору єдиної математичної моделі. Користувач виконує велику кількість коротких вправ, проте між ними відсутня спільна структура ігрового поля, послідовності станів і стратегічної взаємодії з алгоритмічним супротивником. Це знижує можливості використання таких систем для дослідження повноцінних дискретних ігрових процесів, де важливу роль відіграє аналіз конфігурацій, прогнозування наслідків ходів та побудова стратегій.

По-четверте, у значній частині продуктів алгоритми супротивника або істотно спрощені, або взагалі відсутні. У багатьох мобільних логічних іграх користувач розв’язує задачі у режимі «одиночної головоломки», де відсутній активний опонент, що приймає рішення на основі аналізу стану. Навіть там, де присутні комп’ютерні супротивники, їх поведінка часто зводиться до випадкового вибору з невеликого набору варіантів чи використання жорстко зашитих сценаріїв. Така реалізація не дає змоги розглядати гру як модель, у якій можна досліджувати алгоритми прийняття рішень, порівнювати стратегії або експериментувати з різними підходами до оцінювання станів.

По-п’яте, у багатьох випадках обмеженими є можливості модифікації правил гри, параметрів ігрового поля та поведінки супротивника з боку користувача або дослідника. Комерційні продукти здебільшого не передбачають доступу до конфігурацій, що визначають розміри поля, набір допустимих значень, спосіб підрахунку очок чи алгоритми вибору ходів. Це знову ж таки унеможливлює використання таких систем як експериментального полігону для дослідження впливу змін у моделі на динаміку гри та її результати.

Серед недоліків, які мають особливе значення можна також відзначити відсутність відкритої, добре структурованої кодової бази, що демонструє реалізацію повного циклу розробки настільної логіко-математичної гри: від формалізації моделі поля до програмування інтерфейсу, підрахунку очок і супротивника. Більшість розглянутих систем надають лише готовий продукт, а не навчальний приклад архітектури з чітко виділеними класами, рівнями абстракції та зрозумілою взаємодією між компонентами.

Таким чином, визначені нами недоліки існуючих рішень, а саме: закритість математичної моделі, обмежені можливості аналізу й модифікації алгоритмів, домінування тренажерних і мікроігрових формату над повноцінними дискретними іграми зі станами та стратегіями які слугують обґрунтуванням доцільності створення окремого програмного продукту

«Математико». Останній орієнтований не лише на ігрове використання, а й на демонстрацію повноцінної архітектури логіко-математичної гри, придатної для вивчення, аналізу та подальшого науково-практичного опрацювання.

2.5. Обґрунтування необхідності створення програмної реалізації гри «Математико».

На підставі проведеного нами аналізу сучасних навчальних програм та логіко-математичних ігор, а також виявлених недоліків існуючих рішень, постає об’єктивна потреба у розробці програмного засобу, який поєднав би чітко формалізовану математичну модель гри, прозору реалізацію алгоритмів і доступну для аналізу архітектуру програмної системи. Логіко-математична гра «Математико» є зручним об’єктом для такої реалізації, оскільки базується на чітких правилах формування ігрового поля, фіксованому наборі чисел та однозначно визначеній процедурі підрахунку очок за рядками, стовпцями та діагоналями. Це дозволяє побудувати завершену дискретну модель гри, у якій можна формально описати множину станів, переходи між ними та критерії оцінювання результатів.

Необхідність створення саме програмної реалізації гри «Математико» зумовлена кількома ключовими міркуваннями.

1. У проаналізованих нами системах домінують або великі комерційні платформи, орієнтовані на масове використання й інтегровані з хмарною інфраструктурою, або веб- і мобільні тренажери, що пропонують набір окремих міні-ігор. У таких продуктах математична складова часто «захована» за ігровими механіками, а внутрішні алгоритми є закритими для користувача. У випадку гри «Математико» одним з головних завдань є створення програмного засобу, в якому правила гри та алгоритми комп’ютерного супротивника

реалізовано у відкритій і структуровано організованій кодовій базі, придатній для подальшого аналізу та модифікації.

2. Важливим є не лише кінцевий ігровий продукт, а й сама процесуальна сторона розробки: проєктування структури даних, вибір архітектурних підходів, реалізація алгоритмів прийняття рішень, організація взаємодії між інтерфейсом та логікою. Існуючі рішення, як правило, не дають можливості студенту або дослідникові розглянути всі ці аспекти в комплексі, оскільки постають у вигляді завершених, але закритих систем. Розробка власного застосунку «Математико» дозволяє повною мірою продемонструвати повний цикл створення програмного продукту: від формалізації вихідної настільної гри та побудови UML-діаграм до реалізації класів і методів, що відповідають за ігрове поле, комп’ютер-суперник, підрахунок очок та відображення стану гри на екрані.

3. Специфіка гри «Математико» зумовлює потребу в явній реалізації алгоритмів аналізу конфігурації поля. Алгоритм комп’ютерного супротивника має враховувати значення в рядках, стовпцях і діагоналях, оцінювати можливі варіанти розміщення чисел і вибирати хід з урахуванням майбутнього підрахунку результату. Такі алгоритми можуть бути реалізовані з різним ступенем складності: від простих евристик до наближених стратегій, що імітують елементи оптимізаційних підходів. Програмна реалізація гри в середовищі C#/WPF із виділенням окремих сервісних класів для логіки комп’ютера та підрахунку очок створює основу для експериментування з різними варіантами алгоритмів, порівняння їх ефективності та подальшого наукового опрацювання.

4. Доцільність створення настільного застосунку, а не чергової веб- або мобільної гри, пояснюється прагненням досягти максимальної прозорості структури коду та простоти розгортання. Локальний застосунок на базі C# та WPF не вимагає серверної інфраструктури, складної системи авторизації чи інтеграції з віддаленими сервісами. Це дозволяє зосередити увагу на

внутрішній архітектурі та алгоритмах, а не на допоміжних сервісах, що є особливо важливим у контексті навчальних і дослідницьких задач у галузі програмної інженерії та розробки ігрових застосунків.

Крім того, створення програмної реалізації гри «Математико» дозволяє сформувати відкриту платформу, яку можна використовувати як базу для подальшого розширення: зміни розмірів ігрового поля, модифікації набору чисел, варіювання правил підрахунку очок, впровадження різних рівнів складності комп’ютерного супротивника. Така гнучкість істотно відрізняє розроблюваний нами застосунок від більшості закритих комерційних продуктів і дає змогу використовувати його як полігон для практичного відпрацювання навичок проєктування, програмування та аналізу логіко-математичних ігор.

Отже, обґрунтування необхідності створення програмної реалізації гри «Математико» ґрунтується на поєднанні кількох факторів: недостатній прозорості й відкритості існуючих систем; потребі у прикладі повноцінної архітектури настільної логіко-математичної гри; наявності чітко формалізованої вихідної моделі та можливості реалізації й дослідження алгоритмів комп’ютерного супротивника.

РОЗДІЛ 3. ТЕОРЕТИЧНА ЧАСТИНА

3.1. Математична модель гри «Математико»

У даному підпункті нами формалізується математична модель гри «Математико» як дискретної скінченної гри з повною інформацією, що реалізується у вигляді настільного комп'ютерного застосунку. Така модель є основою для подальшого побудування алгоритмів, архітектури програмної системи та аналізу роботи комп'ютера-суперника.

Ігрове поле розглядається як квадратна матриця розміру 5×5. Позначимо через

$$A=(a_{ij}),\quad i,j=1,\ldots,5,$$

де кожен елемент a_{ij} може набувати значення з множини $V=\{1,2,\ldots,13\}\cup\{\emptyset\}$, де $\emptyset$ позначає порожню клітинку (у програмній реалізації - відсутність числа в кнопці інтерфейсу). Таким чином, у будь-який момент часу стан ігрової картки можна описати як конфігурацію матриці A, у якій частина елементів заповнена числами, а частина лишається порожньою.

Набір чисел, що використовується у грі, є скінченним мультисетом. У класичному варіанті, реалізованому в програмі, використовується множина значень 1,…,13 кожне з яких присутнє в однаковій кількості копій (наприклад, по чотири екземпляри кожного числа). Умовно можна задати мультисет

$M=\{1^{(k)},2^{(k)},\ldots,13^{(k)}\}$, де k – кількість копій кожного значення. У процесі гри числам із цього мультисета послідовно відповідають ходи гравця та комп'ютера, причому кожне число може бути використане лише фіксовану кількість разів. У програмній реалізації значення поточного числа передається в логіку гри як параметр поточного ходу.

Стан гри на певному кроці можна описати кортежем

$$S=(A,M',p),$$

де A – поточна конфігурація ігрового поля,

$M' \subseteq M$ підмножина (точніше, підмультисет) ще невикористаних чисел,

а $p \in$ {гравець,комп'ютер} - індикатор того, кому належить наступний хід. Перехід від стану S_t до стану S_{t+1} задається операцією розміщення поточного числа $v \in M'$ у одну з вільних клітинок матриці:

$$a_{ij}=\emptyset \Rightarrow a_{ij}:=v,$$

після чого відповідний елемент видаляється з мультисета M', а індикатор гравця перемикається. У програмному коді ця операція реалізується через перевірку доступності клітинки та встановлення значення контенту відповідної кнопки.

Завершений стан гри відповідає повністю заповненому полю, тобто такому S^*, для якого $\forall i,j \quad a_{ij} \in \{1,\ldots,13\}$ та мультисет невикористаних чисел порожній. Для цього стану визначається функція підрахунку очок:

$F(A)=Fdiag(A)+Frow(A)+Fcol(A)$,

де Fdiag, Frow, Fcol - внески від діагоналей, рядків і стовпців відповідно. У програмній реалізації цю функцію інкапсульовано в окремому сервісному класі, який послідовно викликає методи підрахунку для діагоналей, рядків і колонок.

Кожен рядок, стовпець або діагональ розглядається як впорядкована п'ятірка значень

$$L=(x_1,x_2,x_3,x_4,x_5), \; x_k \in \{1,\ldots,13\}.$$

Для відповідної п'ятірки будується словник кількостей, який задає мультисет значень у лінії:

$$N_L=\{(v,c_v)\},$$

де c_v кількість входжень значення v у лінії. Далі для кожної лінії обчислюється часткова оцінка f(L), що залежить від комбінацій чисел у цій п'ятірці. У загальному вигляді це можна описати як

$$f(L)=f(N_L, \text{тип лінії}),$$

де «тип лінії» (рядок, стовпець чи діагональ) впливає на базовий внесок у суму (наприклад, для діагоналей задається додатковий фіксований бонус).

У реалізованому варіанті гри нами використано комбіновану систему оцінювання, що враховує:

– наявність спеціальних комбінацій (наприклад, чотири однакові одиниці; одночасна присутність певного набору «особливих» чисел; комбінації значень 1 і 13 із фіксованою кратністю);

– утворення послідовностей із п’яти послідовних чисел (умова max(L)−minL)=4 за відсутності повторень);

– кількість різних значень у лінії (наприклад, дві, три, п’ять різних чисел);

– наявність пар або трійок однакових значень (c_v=2,3,4 тощо).

Функція f(L) має вигляд набору умов «якщо–то», де для певних комбінацій додаються відповідні бали. Для діагоналей додається додатковий початковий бонус, що відображає їх підвищену вагу в загальній оцінці. Сукупний результат для картки гравця чи комп’ютера

$$F(A)=\sum_{\text{УСІ ЛІНІЇ } L} f(L)$$

- визначає підсумкову кількість очок, яка використовується для порівняння результатів та визначення переможця.

Таким чином, з математичної точки зору гра «Математико» описується як процес послідовного заповнення матриці 5×5 значеннями з фіксованого мультисета з подальшим обчисленням цільової функції F(A), що є сумою оцінок для всіх рядків, стовпців і діагоналей. Гравці (людина та комп’ютер) здійснюють ходи, обираючи позицію для поточного числа, маючи за мету максимізувати значення своєї функції F. Запропонована формалізація є базою для подальшого аналізу логіко-комбінаторних властивостей гри, побудови алгоритмів супротивника та моделювання програмної системи.

3.2. Логіко-комбінаторні аспекти гри «Математико»

Логіко-комбінаторні аспекти гри «Математико» безпосередньо пов’язані зі структурою ігрового поля, множиною допустимих значень та правилами підрахунку очок. З формальної точки зору гра реалізує типову ситуацію вибору послідовності ходів у скінченному просторі станів, де кожен стан визначається конфігурацією матриці розміру 5×5, а переходи між станами задаються операціями розміщення чисел із фіксованого мультисета. У такій постановці кожен хід гравця або комп’ютера не лише локально змінює одне значення в полі, а й впливає на глобальну оцінку, оскільки окремі розміщення можуть формувати або руйнувати вигідні комбінаторні структури в рядках, стовпцях і діагоналях.

Комбінаторна складність гри зумовлена тим, що навіть за фіксованого розміру поля кількість можливих конфігурацій зростає експоненційно зі збільшенням числа заповнених клітин. На практиці простір станів обмежується конкретним сценарієм роздачі чисел і послідовністю ходів, однак навіть у цьому разі гравець стикається з необхідністю вибору з великої кількості допустимих розміщень. Комбінаторика проявляється, зокрема, у формуванні комбінацій типу «послідовність п’яти чисел», «пара», «трійка», «чотири однакові значення», а також у поєднанні різних множин значень в одній лінії. Кожна з таких структур має власну «вартість» у термінах очок, яку задає функція оцінювання лінії, і саме це створює нетривіальні логіко-комбінаторні ситуації.

З погляду теорії дискретних структур кожен рядок, стовпець або діагональ можна розглядати як мультисет фіксованої потужності. Далі над цим мультисетом аналізуються різні властивості: кількість різних значень, наявність повторень, диапазон значень, що дозволяє виявляти послідовності, комбінації з фіксованою кратністю окремих чисел тощо. У реалізованій нами системі алгоритми підрахунку очок використовують саме такі комбінаторні характеристики: будуються відображення, які для кожного значення фіксують кількість входжень, далі перевіряються умови наявності певних шаблонів

(наприклад, п’ять послідовних значень, чотири одиниці, комбінація «1 та 13» з певною кількістю повторів тощо). Це дозволяє сформувати гнучку систему оцінок, що враховує як локальні, так і глобальні властивості числових наборів.

Важливим логічним аспектом є вибір стратегії розміщення числа з урахуванням можливих майбутніх комбінацій. На відміну від статичних задач, де оцінюється вже готова конфігурація, у грі «Математико» гравець та комп’ютер послідовно формують її, і кожний хід може або наблизити до вигідної комбінації, або, навпаки, заблокувати власні майбутні можливості. Логічний аналіз полягає у виборі таких позицій, які максимізують очікувану кількість очок з урахуванням того, що одна й та ж клітинка може одночасно належати рядку, стовпцю і діагоналі. Таким чином, значущими є клітинки, що розташовані в перетинах «вигідних» ліній, а також клітинки діагоналей, які в моделі отримують додаткову вагу.

Окремої уваги заслуговують симетрії поля. Оскільки матриця 5×5 має осі симетрії (горизонтальну, вертикальну, діагоналі), то певні конфігурації фактично еквівалентні з точку зору оцінювання, хоча геометрично відрізняються. Це створює можливості для оптимізації алгоритмів аналізу, але одночасно ускладнює прогнозування для користувача: деякі ходи, які виглядають різними, на рівні оцінки можуть бути еквівалентними. У реалізованій нами програмі ці симетрії поки що не використовуються для формального скорочення простору станів, однак сама структура гри стимулює їх врахування в можливих подальших удосконаленнях алгоритмів.

Отже, логіко-комбінаторні аспекти гри «Математико» проявляються у виборі послідовності ходів у скінченному, але складному просторі конфігурацій, у формуванні різноманітних комбінацій значень у лініях і в аналізі їх впливу на підсумковий результат. Ці властивості роблять гру зручною моделлю для дослідження стратегій, побудови евристичних алгоритмів і відпрацювання навичок роботи з дискретними структурами в рамках комп’ютерних наук.

3.3. Алгоритми роботи комп’ютера-суперника

Алгоритми роботи комп’ютера-суперника в грі «Математико» є ключовим елементом програмної реалізації, оскільки саме вони визначають рівень «інтелектуальності» поведінки опонента та створюють для користувача відчуття змагання зі стратегічно діючим супротивником. Нами було реалізовано алгоритмічну схему, що поєднує евристичний аналіз стану поля з контролем коректності ходів і резервною стратегією випадкового вибору у ситуаціях, де кілька варіантів мають однакову оцінку.

У загальному вигляді алгоритм роботи комп’ютера для кожного ходу можна описати послідовністю кроків. На вхід передається: поточний стан ігрового поля у вигляді матриці значень, номер або значення числа, яке необхідно розмістити, та інформація про вже зайняті клітинки. Спочатку формується множина кандидатів для розміщення – усі вільні клітинки (i,j), для яких aij=∅. Далі для кожного кандидата виконується оцінювання: віртуально розміщується поточне число у клітинці (i,j), після чого викликаються процедури підрахунку очок для змінених ліній (рядків, стовпців, а за потреби й діагоналей). У реалізації це здійснюється шляхом тимчасової модифікації копії ігрової матриці та передачі її до сервісу, що відповідає за підрахунок результату.

Для кожної потенційної позиції обчислюється локальна оцінка, яка може включати як додаткові очки, отримані в результаті цього ходу, так і бонуси за утворення певних комбінацій у діагоналях або рядках. Після аналізу всіх кандидатів обирається клітинка з максимальною оцінкою. У випадку, коли кілька клітинок дають однаковий результат, використовується додаткова евристика (наприклад, пріоритет діагоналей або центральних клітинок) або застосовується випадковий вибір серед рівноцінних варіантів. Така схема

дозволяє комп’ютеру уникнути повністю детермінованої поведінки в ситуаціях, де існує багато еквівалентних ходів, і водночас забезпечує прагнення до максимізації підсумкової кількості очок.

У реалізованому застосунку алгоритм комп’ютера організовано у вигляді окремого сервісного класу (умовно MatematicoComputer), який інкапсулює всі операції з аналізу поля та вибору координат наступного ходу. Взаємодія з цим класом побудована таким чином, що графічний інтерфейс (головне вікно та користувацькі елементи керування) передає йому тільки необхідний мінімум інформації: поточний стан поля та значення числа для розміщення. У відповідь сервіс повертає пару індексів (i,j), які вказують на обрану клітинку; подальше відображення цього ходу в інтерфейсі виконується на рівні UI-компонентів.

Слід зауважити, що у поточній версії алгоритм комп’ютера орієнтований переважно на максимізацію власного результату, а не на блокування потенційно вигідних ходів користувача. Тобто оцінювання можливих позицій виконується по відношенню до власного ігрового поля комп’ютера. Такий підхід є доцільним на першому етапі розробки та забезпечує достатній рівень складності для базових режимів гри. Разом з тим, побудована нами архітектура дозволяє у подальшому розширити алгоритм, включивши в нього аналіз «дзеркальної» ситуації на полі користувача для вибору ходів, які не тільки збільшують власний результат, а й потенційно зменшують можливості опонента.

З програмно-інженерної точки зору важливою перевагою обраного підходу є чітке розмежування відповідальності: алгоритм комп’ютера не звертається безпосередньо до елементів інтерфейсу, а працює з абстрактним представленням стану поля, що підвищує тестованість і спрощує верифікацію правильності рішень. Така організація коду дає змогу окремо модифікувати й удосконалювати стратегію супротивника, не змінюючи основної логіки інтерфейсу та структури проєкту.

3.4. Моделювання структури програмної системи та UML-діаграми

Моделювання структури програмної системи «Математико» нами здійснювалося із використанням нотації UML, що дало змогу формалізувати як функціональні можливості застосунку, так і внутрішню архітектуру та динаміку взаємодії між його складовими. Такий підхід є методично доцільним для спеціальності «Комп'ютерні науки», оскільки дозволяє перейти від неформального опису гри до строгої моделі, яка може бути реалізована у вигляді програмного коду. У межах роботи були побудовані, зокрема, діаграма класів та діаграма послідовності, що відображають різні рівні абстракції програмної системи (наведені на рис. 3.1, 3.2).

На першому етапі нами було змодельовано зовнішню функціональність системи у вигляді діаграми послідовностей (рис. 3.1). Діаграма послідовностей, відображає послідовність дій та взаємодій між основними компонентами системи під час процесу гри.

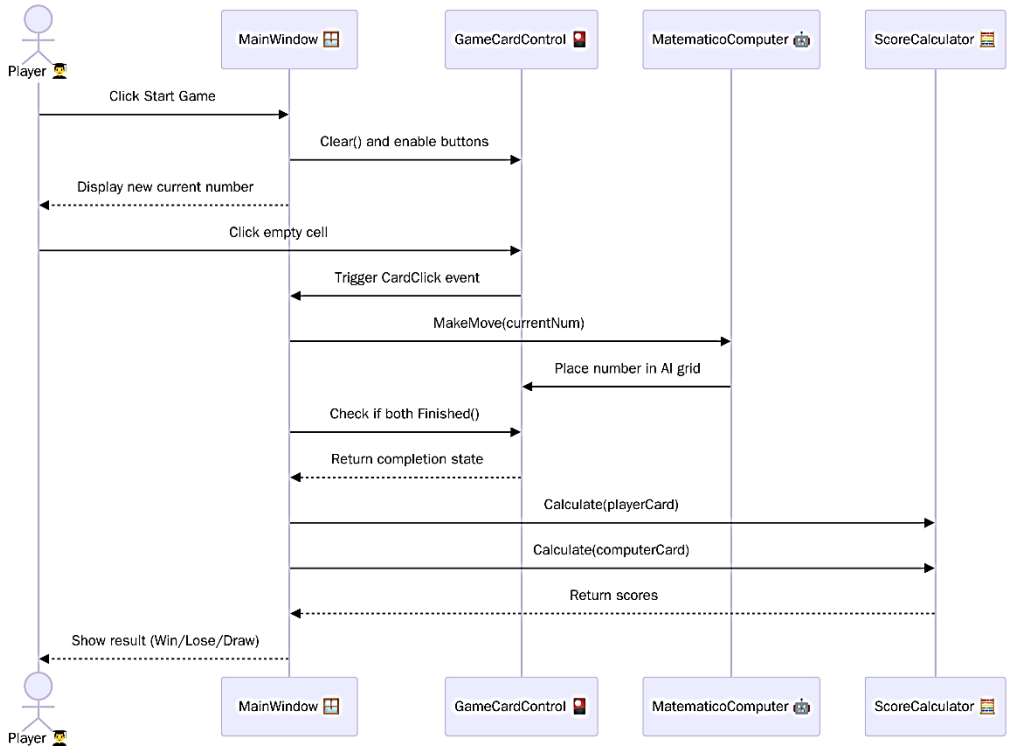

Рисунок 3.1. Діаграма послідовностей гри «Математико»

У межах реалізації на платформі WPF головний об’єкт управління, клас MainWindow, виконує роль координатора взаємодій між користувачем, елементами інтерфейсу, комп’ютерним супротивником та системою обчислення результатів. Початок сценарію ініціюється користувачем, який за допомогою елемента інтерфейсу ButtonStartGame активує подію ButtonStartGame_Click. Ця подія викликає метод очищення і підготовки ігрових полів у контролах типу GameCardControl, після чого гравцеві демонструється нове значення поточної карти, сформоване генератором випадкових чисел.

Після запуску гри користувач взаємодіє з ігровим полем через натискання на окремі клітинки користувацького елемента GameCardControl. У відповідь на подію натискання відбувається виклик методу CardClick у класі MainWindow, який фіксує вибране значення карти, оновлює кількість карт, що залишилися, та ініціює чергу комп’ютера. На цьому етапі викликається метод MakeMove класу MatematicoComputer, у якому реалізовано логіку вибору оптимального місця для розміщення карти комп’ютером. Вибір базується на аналізі поточного стану сітки карток, зокрема перевіряються діагоналі, рядки та стовпці на предмет можливих комбінацій чисел. Якщо жоден зі стратегічних варіантів не підходить, використовується випадкове розташування елемента.

Після кожного ходу системою виконується перевірка завершення гри шляхом виклику методу Finished() у контролях GameCardControl для обох учасників. Якщо всі клітинки заповнені, відбувається підрахунок очок за допомогою статичного класу ScoreCalculator. Цей компонент аналізує комбінації чисел у рядках, стовпцях і діагоналях, присвоюючи бали відповідно до встановлених закономірностей. Результати обчислень повертаються до головного вікна, де вони відображаються користувачу через відповідні елементи інтерфейсу - мітки LabelPlayerPoints і LabelComputerPoints. Завершальним етапом є відображення результату гри за допомогою діалогового вікна, що повідомляє користувача про перемогу, поразку або нічию.

Таким чином, діаграма послідовностей демонструє чітку логіку

послідовного обміну повідомленнями між основними об’єктами програми. Вона відображає циклічність взаємодій між користувачем та системою, де MainWindow виступає посередником між подіями інтерфейсу, штучним інтелектом та модулем підрахунку результатів. Застосована архітектура забезпечує модульність, що дозволяє ізолювати логіку прийняття рішень комп’ютера, обробку подій користувача та механізм підрахунку очок у незалежні компоненти. Такий підхід не лише підвищує гнучкість і розширюваність коду, але й відображає принципи об’єктно-орієнтованого проектування, спрямовані на зменшення зв’язності між класами та підвищення керованості програмного процесу.

На наступному етапі нами було побудовано діаграму класів, яка відображає внутрішню архітектуру програмного продукту (рис. 3.2 - діаграма класів програмної системи «Математико»). Центральне місце у структурі займає клас MainWindow, що відповідає за головне вікно застосунку та координує взаємодію між інтерфейсними і сервісними компонентами. MainWindow містить посилання на об’єкт типу GameCardControl, який інкапсулює подання ігрового поля, а також асоціації з класами MatematicoComputer та ScoreCalculator, що реалізують, відповідно, алгоритми комп’ютера-суперника та підрахунок очок.

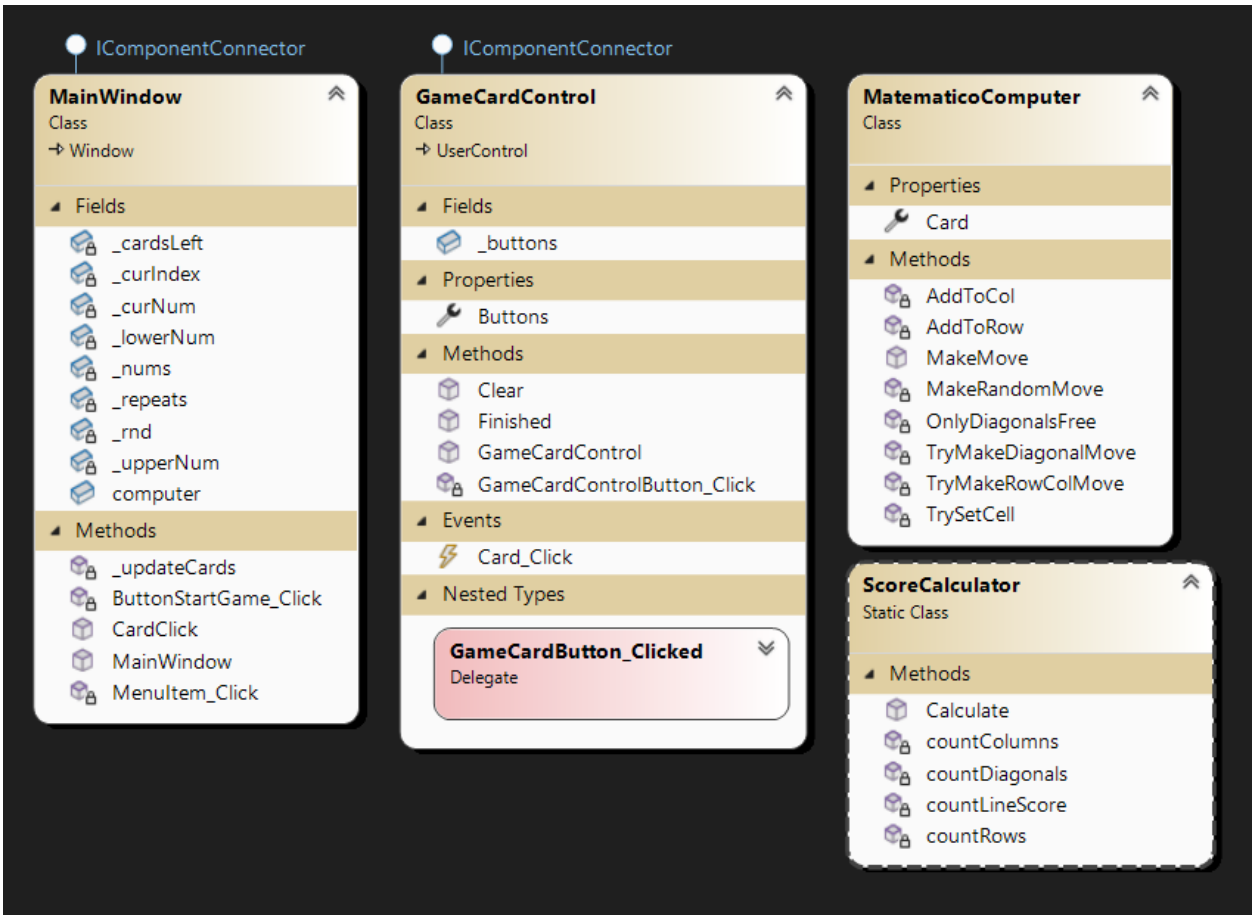

Рисунок 3.2. Діаграма класів застосунку

Користувацький елемент GameCardControl розглянуто нами як окремий клас, який відповідає за відображення матриці 5×5 клітинок та опрацювання подій, пов’язаних із натисканням на них. На діаграмі класів цей компонент має власні атрибути, що описують внутрішню структуру поля (наприклад, масив значень або колекцію об’єктів, які представляють окремі клітинки), та операції для оновлення відображення після кожного ходу. Водночас логіка перевірки допустимості ходів та зміни стану поля реалізується у тісній взаємодії GameCardControl з MainWindow, що фіксується відповідними асоціаціями на діаграмі.

Клас MatematicoComputer на UML-діаграмі представлено як сервісний компонент, який не взаємодіє безпосередньо з елементами інтерфейсу. Нами було задано інтерфейс цього класу у вигляді методів, що приймають як параметри поточний стан поля та значення числа для розміщення і повертають

координати обраної клітинки. Таке інкапсульоване подання алгоритму комп’ютера-суперника дозволяє розглядати його як окремий модуль, який за потреби може бути замінено або розширено без зміни інших частин системи. Аналогічний підхід застосовано до класу ScoreCalculator, який інкапсулює операції підрахунку очок за рядками, стовпцями та діагоналями на основі матриці значень.

Важливою особливістю побудованих -діаграм є те, що вони відображають розподіл відповідальності між компонентами, який нами був закладений на етапі проєктування. Інтерфейсні класи (App, MainWindow, GameCardControl) відповідають за представлення та взаємодію з користувачем, тоді як MatematicoComputer і ScoreCalculator реалізують доменну логіку гри. Такий поділ відображено в діаграмі класів через різні типи зв’язків: композицію між MainWindow та GameCardControl, асоціації між MainWindow і сервісними класами, а також використання примітивних типів і колекцій для зберігання стану поля.

Таким чином, змодельована нами структура програмної системи дозволяє формально й наочно зафіксувати архітектурні рішення, прийняті під час розробки гри «Математико». Діаграма класів описує статичну структуру системи та відношення між її основними компонентами, а діаграма послідовності демонструє динамічний аспект роботи програми під час виконання ключових сценаріїв. У сукупності ці моделі створюють цілісне уявлення про програмну систему та слугують основою для подальшого аналізу, верифікації та модернізації реалізації.

РОЗДІЛ 4. ПРАКТИЧНА ЧАСТИНА

4.1. Обґрунтування вибору технологій реалізації.

Вибір технологій реалізації програмного комплексу для гри «Математико» є одним із ключових етапів проєктування, оскільки саме від нього залежать архітектурні рішення, зручність реалізації алгоритмів, якість графічного інтерфейсу та подальші можливості супроводу системи. У даній роботі нами було прийнято рішення реалізувати застосунок як настільну програму під операційну систему Windows із використанням мови C#, платформи .NET та технології Windows Presentation Foundation (WPF). Такий стек технологій забезпечує поєднання достатньо високого рівня абстракції, зручних інструментів розробки та можливості чіткого розділення інтерфейсної й логічної складових [1-8, 10, 11, 13].

Першою передумовою такого вибору є характер задачі. Гра «Математико» орієнтована на локальне використання як навчально-демонстраційний приклад, що дозволяє вивчати структуру коду, алгоритми та архітектуру програми без залучення мережевої інфраструктури. Для цього найбільш природною є модель автономного настільного застосунку, який не потребує веб-сервера, бази даних у хмарі та складних механізмів розгортання. Платформа .NET історично є однією з базових для розроблення десктопних програм під Windows, а мова C# поєднує виразність сучасних об’єктно-орієнтованих мов із наявністю розвиненої стандартної бібліотеки та засобів підтримки різних типів застосунків.

Другою важливою причиною є наявність у WPF потужних засобів побудови графічного інтерфейсу з використанням декларативної мови розмітки XAML. У межах гри «Математико» інтерфейс складається з ігрового поля 5×5, окремих клітинок, панелей керування, полів для виведення поточного рахунку та службової інформації. Вибір WPF дозволяє описати ці елементи у вигляді

ієрархії візуальних компонентів, задавати стилі та шаблони відображення, створювати власні користувацькі елементи (зокрема елемент GameCardControl для подання ігрової картки) і прив’язувати їх до внутрішньої моделі даних. Завдяки цьому досягається ясне розмежування між описом зовнішнього вигляду (XAML) і реалізацією поведінки (код на C#), що є методично важливим з точки зору навчання принципів проектування інтерфейсів [1, 4, 6, 7, 11].

Третьою причиною на користь обраного стеку є підтримка об’єктно-орієнтованої архітектури та зручність інкапсуляції алгоритмів у вигляді окремих класів і сервісів. Алгоритми підрахунку очок, аналізу рядків, стовпців і діагоналей, а також алгоритм комп’ютера-суперника вимагають оперування з матрицями, словниками частот, множинами можливих ходів. Мова C# надає зручні колекції, лямбда-вирази, LINQ-запити, що робить реалізацію таких алгоритмів компактною та читабельною. Крім того, у межах платформи .NET легко організувати модульну структуру проєкту, де логіка гри виділена в окремі класи (MatematicoComputer, ScoreCalculator тощо), які не залежать від конкретної реалізації інтерфейсу. Це підвищує тестованість коду: окремі модулі можуть перевірятися за допомогою модульних тестів без «підняття» всієї графічної оболонки [1, 4, 7].

Окремо слід зазначити роль інтегрованого середовища розробки Visual Studio, яке є стандартним інструментом для створення застосунків на C# / WPF. Використання цього середовища спрощує налагодження, профілювання та рефакторинг проєкту, а також надає вбудовані засоби для роботи з XAML-розміткою, конструктор форм, засоби перегляду ієрархії елементів інтерфейсу та вікно виведення помилок прив’язки даних. Для навчальних цілей це є вагомою перевагою, оскільки студенти отримують можливість працювати з промисловим інструментарієм, застосовуючи його до відносно компактного, але концептуально завершеного прикладу – гри «Математико».

У ході аналізу альтернатив нами розглядалася можливість реалізації гри

як веб-застосунку із використанням JavaScript-фреймворків (React, Angular) або як мобільного додатка на базі Android (Kotlin/Java) чи кросплатформених рішень (Flutter, React Native). Однак у цих випадках значна частина зусиль була б спрямована на налаштування інфраструктури проєкту, особливостей розгортання, адаптації інтерфейсу до різних розмірів екранів, інтеграції з браузером або мобільною платформою. Для цілей даної дипломної роботи, де основний акцент зроблено на математичній моделі гри, алгоритмах і архітектурі, подібні витрати ресурсів були б методично невиправданими.

Розглядався також варіант використання ігрових рушіїв (Unity, Godot), які забезпечують розширені можливості роботи з графікою, анімаціями та фізикою. Проте для логіко-математичної гри з дискретним полем 5×5 і відсутністю складних графічних ефектів введення повноцінного рушія створює надмірне ускладнення структури проєкту. У цьому разі основним об’єктом вивчення стає не стільки модель гри та алгоритми, скільки сама екосистема рушія. Обраний підхід на основі WPF дозволяє уникнути цього, зосередившись на сутності алгоритмічного та архітектурного рівнів.

Важливим аргументом на користь використання C# / .NET / WPF є також їх поширеність у промисловій розробці десктопних та корпоративних систем. Це означає, що навички, набуті під час реалізації й аналізу гри «Математико», є безпосередньо переносними на реальні задачі в галузі розробки програмного забезпечення: проєктування багатошарових застосунків, робота з подієвою моделлю, створення користувацьких елементів керування, організація взаємодії між інтерфейсом і бізнес-логікою.

Отже, обраний технологічний стек це мова C#, платформа .NET та технологія WPF у поєднанні з середовищем Visual Studio – є обґрунтованим як із позицій технічної доцільності, так і з методичної точки зору. Він забезпечує можливість створення структурованого, розширюваного та наочно організованого програмного продукту, у якому реалізовано логіко-математичну гру «Математико», а також дозволяє використовувати отриманий застосунок як

навчальний приклад для формування математичних та професійних компетентностей у галузі комп’ютерних наук.

4.2. Структура програмного продукту, реалізація ігрового поля та механізму ходів

Структура програмного продукту «Математико» побудована за принципом чіткого розділення відповідальності між інтерфейсною частиною, логікою роботи з ігровим полем та сервісами, що реалізують алгоритми гри. На рівні проєкту це відповідає типовій WPF-аплікації на C#, яка містить точку входу App, головне вікно MainWindow та окремий користувацький елемент керування GameCardControl для відображення ігрової картки. Логіка комп’ютера-суперника та механізм підрахунку очок виділені у сервісні класи, що не залежать від конкретної реалізації інтерфейсу.

Ігрове поле моделюється як квадратна матриця розміром 5×5, що на рівні WPF-інтерфейсу відображається сіткою кнопок. У XAML-розмітці GameCardControl це реалізовано через елемент Grid з п’ятьма рядками та п’ятьма стовпцями, у які програмно або декларативно розміщуються кнопки, що відповідають окремим клітинкам картки. Узагальнений фрагмент XAML-розмітки може мати такий вигляд:

```
<UserControl x:Class="Matematico.GameCardControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid x:Name="CardGrid">
    <Grid.RowDefinitions>
      <RowDefinition Height="*" />
      <RowDefinition Height="*" />
      <RowDefinition Height="*" />
```

```
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <!-- Кнопки клітинок 5×5 створюються під час ініціалізації в code-behind -->
    </Grid>
</UserControl>
```

У цьому фрагменті задається контейнер для таблиці 5×5; самі кнопки створюються в коді й прив’язуються до внутрішньої моделі даних. Така організація дозволяє гнучко керувати створенням клітинок, їх нумерацією та обробкою подій.

Внутрішнє подання ігрового поля інкапсульовано в класі GameCardControl. Для цього використовується двовимірний масив або таблиця nullable-значень (наприклад, int?[,] cells), що відображає поточний стан картки: число в клітинці або відсутність значення. Узагальнений фрагмент класу можна подати так:

```
public partial class GameCardControl : UserControl
{
    private readonly int?[,] _cells = new int?[5, 5];

    public event Action<int, int>? CellClicked;
```

```
public GameCardControl()
{
    InitializeComponent();
    InitializeGrid();
}

private void InitializeGrid()
{
    for (int i = 0; i < 5; i++)
    for (int j = 0; j < 5; j++)
    {
        var button = new Button();
        button.Tag = (i, j);
        button.Click += OnCellButtonClick;
        Grid.SetRow(button, i);
        Grid.SetColumn(button, j);
        CardGrid.Children.Add(button);
    }
}

private void OnCellButtonClick(object sender, RoutedEventArgs e)
{
    var (i, j) = ((int, int))((Button)sender).Tag;
    CellClicked?.Invoke(i, j);
}

public bool TrySetCell(int i, int j, int value)
{
    if (_cells[i, j].HasValue)
```

```
            return false;

        _cells[i, j] = value;

        foreach (var child in CardGrid.Children.OfType<Button>())
        {
            var (row, col) = ((int, int))child.Tag;
            if (row == i && col == j)
            {
                child.Content = value.ToString();
                break;
            }
        }

        return true;
    }

    public int?[,] GetBoardState()
    {
        return (int?[,])_cells.Clone();
    }
}
```

У наведеному коді реалізовано:

– створення решітки кнопок і прив’язування до них обробника кліку;

– подію CellClicked, яка інформує головне вікно про натискання на клітинку;

– метод TrySetCell, що перевіряє, чи вільна комірка, оновлює внутрішню матрицю _cells та відображення в інтерфейсі;

– метод GetBoardState, який повертає копію поточного стану поля для

подальшого аналізу.

Механізм ходів організовано на рівні класу MainWindow, який координує взаємодію між користувачем, ігровим полем і сервісами логіки. У конструкторі головного вікна підписується обробник на подію CellClicked з боку GameCardControl, а також ініціалізуються об’єкти, що відповідають за комп’ютера-суперника та підрахунок очок. Узагальнений фрагмент може виглядати так:

```
public partial class MainWindow : Window
{
    private readonly MatematicoComputer _computer = new MatematicoComputer();
    private readonly ScoreCalculator _scoreCalculator = new ScoreCalculator();
    private int _currentValue;

    public MainWindow()
    {
        InitializeComponent();
        GameCard.CellClicked += OnPlayerCellClicked;
        StartNewGame();
    }

    private void StartNewGame()
    {
        _currentValue = 1;
        // Скидання стану ігрового поля, рахунку, черги ходів тощо
    }

    private void OnPlayerCellClicked(int i, int j)
    {
```

```
            if (!GameCard.TrySetCell(i, j, _currentValue))
                return; // комірка зайнята

            UpdateScores();

            // Хід комп'ютера
            var board = GameCard.GetBoardState();
            var (ci, cj) = _computer.ChooseMove(board, _currentValue);
            GameCard.TrySetCell(ci, cj, _currentValue);

            UpdateScores();
            _currentValue = GetNextValue();
        }

        private void UpdateScores()
        {
            var board = GameCard.GetBoardState();
            int score = _scoreCalculator.CalculateTotal(board);
            PlayerScoreTextBlock.Text = score.ToString();
            // Аналогічно — рахунок комп'ютера, якщо використовується окрема
картка
        }

        private int GetNextValue()
        {
            // Отримання наступного числа з набору 1..13 з урахуванням
кратності
            // (наприклад, через індекс у списку або генератор послідовності)
            return _currentValue + 1;
```

```
    }
}
```

У цьому фрагменті показано:

– метод OnPlayerCellClicked, який викликається при натисканні на клітинку;

– логіку перевірки допустимості ходу через TrySetCell;

– виклик сервісу підрахунку очок ScoreCalculator через метод UpdateScores;

– організацію ходу комп’ютера через _computer.ChooseMove з передаванням поточного стану поля;

– зміну поточного значення _currentValue, яке використовується для наступних ходів.

Такий механізм забезпечує єдину точку керування ігровим процесом у MainWindow: усі кліки з поля надходять сюди, тут же здійснюється перевірка, оновлення інтерфейсу та виклик необхідних сервісів.

Алгоритм вибору ходу комп’ютером реалізовано в окремому сервісному класі MatematicoComputer, який працює лише з абстрактним поданням поля (двовимірний масив значень) і не залежить від інтерфейсних елементів. Узагальнений вигляд методу може бути таким:

```
public class MatematicoComputer
{
    public (int i, int j) ChooseMove(int?[,] board, int value)
    {
        (int i, int j)? best = null;
        int bestScore = int.MinValue;

        for (int r = 0; r < 5; r++)
        for (int c = 0; c < 5; c++)
        {
```

```
            if (board[r, c].HasValue) continue;

            board[r, c] = value;
            int score = EvaluatePosition(board);
            board[r, c] = null;

            if (score > bestScore)
            {
                bestScore = score;
                best = (r, c);
            }
        }

        return best ?? (0, 0);
    }

    private int EvaluatePosition(int?[,] board)
    {
        // Виклик ScoreCalculator або спрощена локальна оцінка
        return 0;
    }
}
```

Цей фрагмент демонструє, що клас перебирає всі вільні клітинки; тимчасово розміщує число, оцінює конфігурацію (через EvaluatePosition); обирає позицію з максимальною оцінкою.

Таким чином, структура програмного продукту організована за модульним принципом: GameCardControl відповідає за відображення ігрового поля та базові операції зі станом клітинок, MainWindow координує ігровий процес і виклики сервісних класів, а MatematicoComputer та ScoreCalculator

реалізують, відповідно, алгоритми вибору ходу й підрахунку результатів. Така побудова забезпечує прозорість механізму ходів, можливість окремого тестування логіки гри та полегшує подальше вдосконалення алгоритмів без суттєвої зміни інтерфейсної частини.

4.3. Реалізація алгоритмів комп’ютера-суперника

Алгоритми комп’ютера-суперника в програмному продукті «Математико» реалізовано як окремий модуль доменної логіки, винесений у спеціальний сервісний клас. Такий підхід дозволяє відокремити механізм прийняття рішень від інтерфейсної частини та структури вікон, що, з одного боку, підвищує прозорість реалізації, а з іншого - спрощує тестування й подальшу модернізацію алгоритмів. Нами було обрано евристичну схему, яка на кожному ході здійснює повний перебір доступних позицій, виконує локальну оцінку кожного можливого ходу з урахуванням поточного стану ігрового поля і обирає той варіант, який дає найкращий прогнозований результат для комп’ютера.

Базовий інтерфейс модуля комп’ютера-суперника реалізовано у вигляді класу MatematicoComputer, який містить публічний метод вибору ходу та одна чи кілька допоміжних процедур оцінювання. Узагальнена сигнатура методу вибору ходу має вигляд:

```
public class MatematicoComputer
{
    public (int i, int j) ChooseMove(int?[,] board, int value)
    {
        // реалізація вибору координат ходу
    }
```

```
    // допоміжні методи оцінювання
}
```

У цьому методі параметр board представляє поточний стан поля як двовимірний масив 5×5 із nullable-значеннями (число в клітинці або null, якщо клітинка порожня), а параметр value — числове значення, яке комп’ютер має розмістити на полі на даному ході. Результатом роботи є пара індексів (i, j), що визначає вибрану клітинку.

Загальна схема роботи алгоритму полягає у виконанні таких кроків:

1. побудова множини всіх вільних клітинок поля;
2. для кожної вільної позиції — віртуальне розміщення числа value та оцінювання отриманої конфігурації;
3. вибір клітинки, для якої значення цільової функції (кількість очок) є максимальним;
4. у випадку наявності кількох рівноцінних варіантів — застосування додаткових евристик або випадкового вибору.

Відповідна реалізація методу ChooseMove може мати такий вигляд:

```
public (int i, int j) ChooseMove(int?[,] board, int value)
{
    (int i, int j)? best = null;
    int bestScore = int.MinValue;

    for (int r = 0; r < 5; r++)
    for (int c = 0; c < 5; c++)
    {
        if (board[r, c].HasValue)
            continue; // клітинка зайнята

        // тимчасове розміщення числа
        board[r, c] = value;
```

```
            int score = EvaluatePosition(board);

            // скасування тимчасового ходу
            board[r, c] = null;

            if (score > bestScore)
            {
                bestScore = score;
                best = (r, c);
            }
        }

        // резервний варіант на випадок, якщо вільних клітинок немає
        return best ?? (0, 0);
    }
```

У наведеному фрагменті реалізовано повний перебір усіх порожніх клітинок; для кожної з них викликається допоміжний метод EvaluatePosition, який обчислює оціночне значення конфігурації. Важливо, що розміщення числа виконується на копії логічного стану, а після оцінювання тимчасова зміна відміняється, що унеможливлює небажані побічні ефекти.

Для оцінювання конфігурації нами передбачено окремий метод, який може реалізовуватися двома основними способами: через виклик вже наявного сервісу підрахунку очок (клас ScoreCalculator), який застосовується і для фінального результату; через локальну «полегшену» функцію, яка аналізує лише ті лінії (рядок, стовпець, діагоналі), що були змінені даним ходом. У більш загальному вигляді метод EvaluatePosition можна подати так:

```
private readonly ScoreCalculator _scoreCalculator = new ScoreCalculator();
```

```
private int EvaluatePosition(int?[,] board)
{
   // Варіант 1: повний перерахунок
   int total = _scoreCalculator.CalculateTotal(board);
   return total;

   // Або варіант 2: локальна оцінка змінених рядків/стовпців/діагоналей
}
```

У першому варіанті алгоритм оцінки спирається на ту саму функцію оцінювання, що й фінальний підрахунок очок у грі. Це забезпечує узгодженість: хід, який максимізує значення CalculateTotal, є «найкращим» з точки зору загальної моделі. У другому варіанті можлива подальша оптимізація через аналіз лише локально змінених ліній, однак такий підхід потребує додаткової логіки та зберігання попередніх проміжних результатів.

Структура класу ScoreCalculator, на який спирається комп’ютер, передбачає метод CalculateTotal, який обчислює суму внесків усіх рядків, стовпців та діагоналей. Узагальнений вигляд методу:

```
public class ScoreCalculator
{
   public int CalculateTotal(int?[,] board)
   {
      int total = 0;

      // рядки
      for (int r = 0; r < 5; r++)
      {
         var line = GetLine(board, r, isRow: true);
         total += CalculateLine(line);
      }
```

```
        // стовпці
        for (int c = 0; c < 5; c++)
        {
            var line = GetLine(board, c, isRow: false);
            total += CalculateLine(line);
        }

        // діагоналі
        var mainDiag = GetMainDiagonal(board);
        var sideDiag = GetSideDiagonal(board);
        total += CalculateLine(mainDiag, isDiagonal: true);
        total += CalculateLine(sideDiag, isDiagonal: true);

        return total;
    }

    // допоміжні методи отримання ліній та підрахунку оцінки
}
```

У цьому методі реалізовано логіку комбінаторної оцінки: аналіз наборів із п’яти чисел, визначення наявності послідовностей, мультисетів з короткими чи довгими повтореннями, особливих комбінацій тощо. Таким чином, комп’ютер-суперник, обираючи хід, фактично прогнозує майбутнє значення цієї функції для власного поля.

Окрему роль відіграють евристики, що застосовуються в разі рівнозначних варіантів. Оскільки в реальній грі часто трапляються ситуації, де кілька можливих ходів дають однаковий результат за значенням EvaluatePosition, нами передбачено можливість вводити додаткові критерії вибору. Зокрема, можуть використовуватися такі правила:

– пріоритет клітинок, які належать діагоналям (через їхню підвищену вагу в моделі підрахунку);

– пріоритет центральної клітинки (0, 0 у матриці 5×5), яка одночасно входить до двох діагоналей;

– випадковий вибір серед множини найкращих за оцінкою позицій для уникнення повністю детермінованої поведінки.

Загальна схема введення такої евристики у код може виглядати як збирання списку «кандидатів з найкращим значенням» і подальший вибір конкретної клітинки з цього списку:

```
public (int i, int j) ChooseMove(int?[,] board, int value)
{
   var bestCells = new List<(int i, int j)>();
   int bestScore = int.MinValue;

   for (int r = 0; r < 5; r++)
   for (int c = 0; c < 5; c++)
   {
      if (board[r, c].HasValue) continue;

      board[r, c] = value;
      int score = EvaluatePosition(board);
      board[r, c] = null;

      if (score > bestScore)
      {
         bestScore = score;
         bestCells.Clear();
         bestCells.Add((r, c));
      }
```

```
            else if (score == bestScore)
            {
                bestCells.Add((r, c));
            }
        }

        // вибір однієї з кращих позицій (наприклад, випадковий)
        var rnd = new Random();
        int index = rnd.Next(bestCells.Count);
        return bestCells[index];
    }
```

Такий підхід забезпечує баланс між «раціональною» (з точки зору кількості очок) та «варіативною» поведінкою комп’ютера, що позитивно позначається на ігровому досвіді.

Важливо наголосити, що у поточній версії гри алгоритм комп’ютера-суперника орієнтований насамперед на максимізацію власного результату на одному ході (однокрокова евристична стратегія). Тобто оцінювання відбувається для поточного кроку без моделювання кількох майбутніх ходів або без врахування можливих реакцій гравця. Така схема є цілком достатньою для базового рівня складності й дозволяє сконцентрувати увагу на коректності реалізації функції оцінювання та структури коду. Водночас обраний спосіб інкапсуляції алгоритму у вигляді окремого класу з методами ChooseMove та EvaluatePosition відкриває можливість подальшого розширення: можна додати багатокроковий пошук (мінімакс, альфа-бета відсікання), окремий режим «блокування» перспективних ліній користувача, параметри рівня складності тощо, не змінюючи інтерфейсних компонентів.

Таким чином, реалізація алгоритмів комп’ютера-суперника в програмному продукті «Математико» базується на чітко визначеній математичній моделі гри та узгодженій функції оцінювання, організована у

вигляді окремого сервісного класу, який працює з абстрактним поданням ігрового поля, і забезпечує прозору, формально описану поведінку опонента, придатну як для ігрового використання, так і для подальшого аналізу в межах дисциплін, пов’язаних із алгоритмами й штучним інтелектом.

4.4. Реалізація підрахунку очок та візуалізація результатів

Підрахунок очок у грі «Математико» реалізовано як окремий модуль доменної логіки, інкапсульований у класі ScoreCalculator. Такий підхід дозволяє відокремити математичну модель оцінювання від інтерфейсної частини та механізму ходів, забезпечуючи прозорість реалізації та можливість незалежного тестування. В основу алгоритмів покладено описану в теоретичній частині математичну модель: сума результатів по всіх рядках, стовпцях і діагоналях ігрового поля, де кожна лінія оцінюється за набором логіко-комбінаторних правил (наявність послідовностей, пар, трійок, чотвірок, спеціальних комбінацій тощо).

На рівні коду клас ScoreCalculator має публічний метод CalculateTotal, який приймає поточний стан поля у вигляді двовимірного масиву int?[,] board і повертає ціле число – сумарну кількість очок, набраних на цій картці:

```
public class ScoreCalculator
{
    public int CalculateTotal(int?[,] board)
    {
        int total = 0;

        // підрахунок по рядках
        for (int r = 0; r < 5; r++)
        {
```

```
            int[] line = GetRow(board, r);
            total += CalculateLine(line, isDiagonal: false);
        }

        // підрахунок по стовпцях
        for (int c = 0; c < 5; c++)
        {
            int[] line = GetColumn(board, c);
            total += CalculateLine(line, isDiagonal: false);
        }

        // підрахунок по діагоналях
        int[] mainDiag = GetMainDiagonal(board);
        int[] sideDiag = GetSideDiagonal(board);
        total += CalculateLine(mainDiag, isDiagonal: true);
        total += CalculateLine(sideDiag, isDiagonal: true);

        return total;
    }
}
```

У цьому фрагменті реалізовано загальну схему оцінювання: послідовний обхід усіх рядків, стовпців і діагоналей поля, витягування відповідних п’ятірок значень та їх передавання в допоміжний метод CalculateLine. Окремі методи GetRow, GetColumn, GetMainDiagonal, GetSideDiagonal проставляють значення в одномірні масиви з урахуванням того, що на полі можуть залишатися порожні клітинки; у таких випадках при підрахунку очок або повертаються нульові значення, або лінія може ігноруватися, залежно від обраних правил. Узагальнений вигляд одного з таких методів:

```
private int[] GetRow(int?[,] board, int rowIndex)
```

```
{
    int[] result = new int[5];
    for (int c = 0; c < 5; c++)
    {
        result[c] = board[rowIndex, c] ?? 0;
    }
    return result;
}
```

Ключовим елементом є метод CalculateLine, який реалізує логіку оцінювання окремої лінії (рядка, стовпця або діагоналі). У ньому реалізовано побудову словника частот значень, визначення кількості різних чисел, аналіз наявності послідовностей та спеціальних комбінацій. Узагальнений варіант такого методу може мати вигляд:

```
private int CalculateLine(int[] values, bool isDiagonal)
{
    // базовий бонус для діагоналей
    int score = isDiagonal ? 5 : 0;

    // словник кількостей
    var counts = new Dictionary<int, int>();
    foreach (int v in values)
    {
        if (v == 0) continue; // порожні клітинки
        if (!counts.ContainsKey(v))
            counts[v] = 0;
        counts[v]++;
    }

    int distinctCount = counts.Count;
```

```
        // приклад: послідовність п’яти послідовних чисел
        var nonZero = values.Where(v => v != 0).Distinct().OrderBy(v => v).ToArray();
        if (nonZero.Length == 5 && nonZero.Max() - nonZero.Min() == 4)
            score += 20;

        // приклад: чотири однакові числа
        if (counts.Values.Any(c => c == 4))
            score += 15;

        // приклад: трійка і пара
        bool hasThree = counts.Values.Any(c => c == 3);
        bool hasTwo = counts.Values.Any(c => c == 2);
        if (hasThree && hasTwo)
            score += 10;

        // приклад: спеціальна комбінація з 1 та 13
        if (counts.ContainsKey(1) && counts.ContainsKey(13))
            score += 5;

        // бонус за велику кількість різних значень
        if (distinctCount == 5)
            score += 5;
        else if (distinctCount == 4)
            score += 3;

        return score;
    }
```

У наведеному прикладі реалізовано типову структуру, яку ми використовуємо в програмному продукті:

– спочатку додається базовий бонус за діагональ (якщо isDiagonal = true);

– будується словник counts, що відображає значення на їх кратність у лінії;

– перевіряється наявність послідовності з п’яти послідовних чисел;

– аналізуються мультисети (чотири однакові, трійка + пара);

– перевіряються спеціальні логіко-комбінаторні умови (наявність 1 і 13 тощо);

– додаються бонуси за кількість різних значень.

Фактичний набір правил може бути розширений або модифікований, але загальна структура залишається сталою: кожна лінія розглядається як об’єкт аналізу, для якого обчислюється частковий внесок у сумарну кількість очок.

Реалізація підрахунку очок інтегрується з інтерфейсом через головне вікно MainWindow. Після кожного ходу користувача або комп’ютера викликається метод, який оновлює числові показники на екрані. Узагальнений фрагмент такого методу було використано в попередньому підпункті, тут наведемо його у контексті візуалізації результатів:

```
private void UpdateScores()
{
    int?[,] playerBoard = PlayerCard.GetBoardState();
    int playerScore = _scoreCalculator.CalculateTotal(playerBoard);
    PlayerScoreTextBlock.Text = playerScore.ToString();

    int?[,] computerBoard = ComputerCard.GetBoardState();
    int computerScore = _scoreCalculator.CalculateTotal(computerBoard);
    ComputerScoreTextBlock.Text = computerScore.ToString();

    HighlightLeader(playerScore, computerScore);
```

```
}
```

У цьому фрагменті реалізовано:

– отримання стану поля гравця й комп’ютера через метод GetBoardState відповідних GameCardControl;

– обчислення сумарного результату для кожної картки через _scoreCalculator.CalculateTotal;

– виведення значень у текстові елементи інтерфейсу PlayerScoreTextBlock та ComputerScoreTextBlock;

– виклик додаткового методу HighlightLeader, який відповідає за візуальне виділення поточного лідера.

Метод HighlightLeader може змінювати, наприклад, колір тла або шрифту об’єктів, що відповідають за відображення рахунку, або оформлення рамки навколо картки:

```
private void HighlightLeader(int playerScore, int computerScore)
{
    if (playerScore > computerScore)
    {
        PlayerScoreBorder.BorderBrush = Brushes.Green;
        ComputerScoreBorder.BorderBrush = Brushes.Gray;
    }
    else if (playerScore < computerScore)
    {
        PlayerScoreBorder.BorderBrush = Brushes.Gray;
        ComputerScoreBorder.BorderBrush = Brushes.Green;
    }
    else
    {
        PlayerScoreBorder.BorderBrush = Brushes.Goldenrod;
        ComputerScoreBorder.BorderBrush = Brushes.Goldenrod;
```

```
    }
}
```

Такий підхід забезпечує наочну візуалізацію результатів: користувач не лише бачить числові значення, а й отримує миттєвий графічний зворотний зв’язок щодо того, хто наразі лідирує в грі. За потреби може бути реалізовано й додаткові елементи візуалізації: відображення історії змін рахунку, підсвічування ліній, що принесли найбільшу кількість очок, тощо.

Таким чином, реалізація підрахунку очок та візуалізація результатів у програмному продукті «Математико» ґрунтуються на чітко формалізованій функції оцінювання, винесеній в окремий сервісний клас ScoreCalculator, та на механізмі регулярного оновлення інтерфейсу через методи головного вікна. Поділ на логічний та візуальний рівні дозволяє – змінювати або ускладнювати правила оцінювання без втручання в інтерфейс; незалежно тестувати правильність підрахунку; гнучко варіювати способи графічного відображення результатів, не змінюючи алгоритмічної частини системи.

Це робить програмний продукт придатним як для демонстрації математичної моделі гри, так і для навчальних цілей у галузі розробки програмного забезпечення.

4.5. Інтерфейс користувача та особливості використання

Програмний продукт «Математико» реалізує інтелектуальну настільну гру з елементами логічного мислення та стратегічного планування, призначену для гри між користувачем та комп’ютером. Основна мета гри полягає у заповненні квадратного поля розміром 5×5 числами від 1 до 13, які генеруються з колоди у 52 карти (по чотири копії кожного числа). У процесі гри користувач і

комп’ютер по черзі розміщують оголошене число на своїх ігрових полях, прагнучи сформувати комбінації, що приносять найбільшу кількість очок. Завершення гри відбувається, коли всі клітинки заповнені, після чого система здійснює підрахунок очок для кожного учасника та визначає переможця.

Алгоритм роботи програми базується на чіткій послідовності взаємодії між основними компонентами: головним вікном застосунку (MainWindow), користувацьким елементом керування (GameCardControl), класом логіки комп’ютера (MatematicoComputer) та модулем обчислення очок (ScoreCalculator). Після натискання користувачем кнопки «Start» у головному вікні ініціалізується нова партія: обнуляються результати, очищуються поля та створюється список чисел для подальшої генерації випадкових карт. Під час кожного ходу головне вікно відображає поточне число, після чого користувач самостійно обирає клітинку для його розміщення. Система перевіряє коректність дії, оновлює стан поля гравця і викликає метод MakeMove() об’єкта MatematicoComputer, який відповідає за хід комп’ютера.

Логіка роботи комп’ютера реалізована як ієрархічний алгоритм прийняття рішення. Першим кроком є спроба виконати стратегічне розміщення числа на діагоналях методом TryMakeDiagonalMove(). Цей етап має пріоритет для так званих «особливих» чисел (1, 10, 11, 12, 13), які мають високу комбінаційну цінність у подальшому підрахунку очок. Програма аналізує головну (зліва направо) і побічну (справа наліво) діагоналі матриці кнопок, і якщо поточне число ще не зустрічається на діагоналі, комп’ютер намагається розмістити його у першій доступній порожній клітинці. Якщо ж діагональні комірки вже заповнені або не відповідають умовам, система переходить до наступного етапу - виклику методу TryMakeRowColMove(), який реалізує складнішу логіку. На цьому етапі комп’ютер аналізує кожен рядок і стовпець, підраховуючи частоти появи чисел і кількість вільних клітин. Далі, за визначеними правилами, він прагне або продовжити існуючу комбінацію, або створити нову послідовність: додає число, якщо воно зустрічається один чи два

рази, або завершує серію з трьох чи чотирьох однакових чисел. Якщо жодна умова не виконується, викликається метод MakeRandomMove(), який розміщує число у випадковій порожній клітинці. Для забезпечення коректності розміщення використовується допоміжний метод TrySetCell(), що перевіряє, чи є клітинка вільною.

Алгоритм ходу комп'ютера (MakeMove)

1. Спроба зайняти діагональ → TryMakeDiagonalMove
2. Якщо це не вдалося → спроба зайняти рядок або стовпець → TryMakeRowColMove
3. Якщо і цей варіант не спрацював → виконується випадковий хід → MakeRandomMove

Діагоналі (TryMakeDiagonalMove)

Комп'ютер збирає значення у матриці кнопок (values).

Перевіряє головну діагональ (зліва зверху направо вниз) та побічну діагональ (справа зверху наліво вниз).

Якщо поточне число (currentNum) належить до «особливих» (1, 10, 11, 12, 13), комп'ютер намагається розмістити його на вільному місці діагоналі, але лише за умови, що на цій діагоналі ще немає такого числа.

Для розміщення числа використовується метод TrySetCell, який гарантує, що комп'ютер не перезапише зайняту клітинку.

Рядки та стовпці (TryMakeRowColMove)

Алгоритм є більш складним.

Для кожного рядка та стовпця комп'ютер підраховує, скільки разів зустрічається кожне число, і визначає кількість порожніх клітинок (позначених як -1).

Далі перевіряються різні правила:

– Якщо в рядку або стовпці вже є одиниця, і поточне число також 1 → ставимо її туди.

– Якщо є число, яке зустрічається тричі, і поточне число таке саме → ставимо його четвертим.

– Якщо є комбінація «3 + 1» або два однакові числа «2 + 2» → додаємо ще одне, щоб продовжити ряд.

– Якщо число зустрічається двічі → ставимо його втретє.

– Якщо число зустрічається один раз → ставимо вдруге.

Як тільки одна з умов виконується, число розміщується у першу вільну клітинку відповідного рядка або стовпця.

Випадковий хід (MakeRandomMove)

Якщо жодна з попередніх стратегій не підходить:

Комп’ютер обирає випадкову клітинку за допомогою генератора випадкових чисел (Random).

Якщо є ще вільні клітинки поза діагоналями — вибирається будь-яка порожня.

Якщо залишилися лише порожні клітинки на діагоналях — вибирається одна з них.

Комп’ютер ніколи не ставить число у зайняту клітинку, оскільки в циклі while перевіряється умова Content == null.

Допоміжні методи

TrySetCell - розміщує число у клітинці лише тоді, коли вона порожня.

AddToRow / AddToCol - шукають першу вільну клітинку в рядку або стовпці та вставляють туди число (також через TrySetCell).

OnlyDiagonalsFree - перевіряє, чи залишилися вільні клітинки тільки на діагоналях.

Підсумкова поведінка

1. Комп’ютер насамперед намагається створити комбінації на діагоналях, використовуючи «особливі» числа.

2. Далі прагне посилити або продовжити ряд чи стовпець, щоб зібрати більшу комбінацію.

3. Якщо жоден стратегічний варіант не доступний - виконує випадковий хід, але завжди у порожню клітинку.

Після виконання ходів обох учасників система здійснює підрахунок очок. Модуль ScoreCalculator аналізує кожен рядок, стовпець та діагоналі заповненого поля, формуючи словники частот чисел і обчислюючи бали згідно з фіксованою шкалою. Наприклад, два однакових числа у рядку приносять 10 очок, а на діагоналі - 20. Три однакових числа оцінюються у 40 або 50 очок відповідно, чотири - у 160 або 170. Комбінації з послідовних чисел, наприклад 9–10–11–12–13, дають 50 очок у рядку або 60 на діагоналі. Окремі набори, що включають три одиниці та дві тринадцятки, або числа 1, 10, 11, 12, 13, оцінюються вищими балами (100–160 очок залежно від розташування). Максимальний бал за чотири одиниці у рядку - 200 очок, а на діагоналі - 210. Після обчислення очок для обох сторін результати порівнюються, і програма виводить повідомлення про перемогу, поразку або нічию.

Таким чином, робота програми «Математико» демонструє комплексну інтеграцію інтерфейсної взаємодії, алгоритмічного аналізу та штучної логіки. Програмна реалізація забезпечує адаптивну поведінку комп’ютера, яка поєднує елементи стратегічного мислення (пріоритет діагоналей, аналіз частот чисел) з імовірнісним підходом (випадкове розміщення). Завдяки чітко визначеному механізму підрахунку очок система створює умови для об’єктивного оцінювання результатів, що робить гру збалансованою, інтелектуальною та придатною для аналітичного вивчення алгоритмів прийняття рішень у штучних системах.

Інтерфейс користувача програмного продукту «Математико» спроєктовано як настільний графічний інтерфейс на базі WPF, орієнтований на простоту сприйняття, структурованість елементів і прозоре відображення внутрішнього стану гри. Основна мета інтерфейсних рішень полягає не в створенні візуально перевантаженого ігрового середовища, а у забезпеченні максимальної відповідності між тим, що користувач бачить на екрані, та

формальною математичною моделлю ігрового поля, алгоритмами підрахунку очок і роботою комп’ютера-суперника.

Головне вікно застосунку реалізовано в класі MainWindow і містить кілька логічних зон. Центральне місце відведено ігровому полю, яке подається у вигляді квадратної таблиці 5×5. Для відображення цієї таблиці використано користувацький елемент GameCardControl, що інкапсулює сітку з 25 кнопок, кожна з яких відповідає окремій клітинці. На початку гри всі клітинки є порожніми; у процесі виконання ходів у кнопках відображаються числові значення, які розміщуються на полі гравцем і комп’ютером. Такий підхід дозволяє користувачу одночасно спостерігати за повною конфігурацією поля й безпосередньо співвідносити розташування чисел із результатами підрахунку.

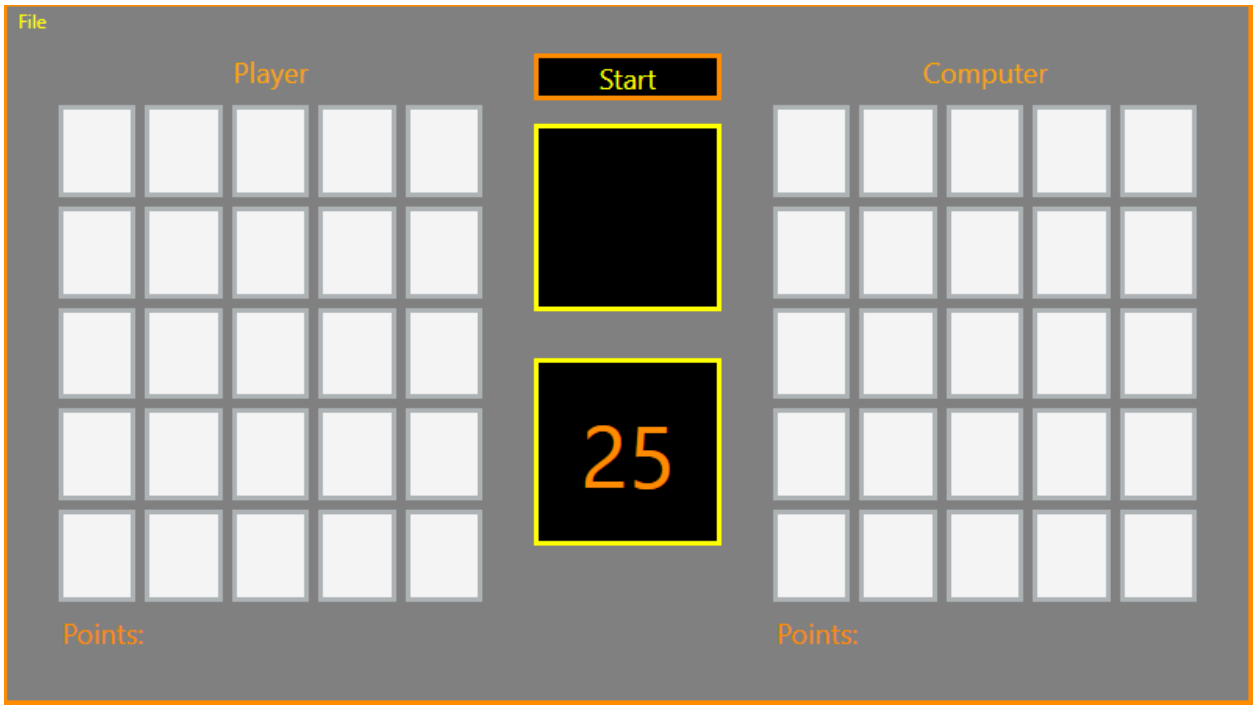

Рисунок 4.1. Інтерфейс застосунку

Окрім ігрового поля, у головному вікні розташовано інформаційний блок, що містить індикатори поточного рахунку, службові повідомлення та керівні елементи. Результати гри (кількість очок гравця та комп’ютера) відображаються у вигляді текстових полів або міток (наприклад,

PlayerScoreTextBlock, ComputerScoreTextBlock), прив’язаних до відповідних значень, які оновлюються після кожного ходу через виклик сервісу підрахунку очок. Розроблено також простий механізм візуального виділення поточного лідера, що реалізується, зокрема, через зміну кольору рамки або фону елементів, пов’язаних із відображенням рахунку. Це створює додатковий візуальний зворотний зв’язок без перевантаження інтерфейсу декоративними елементами.

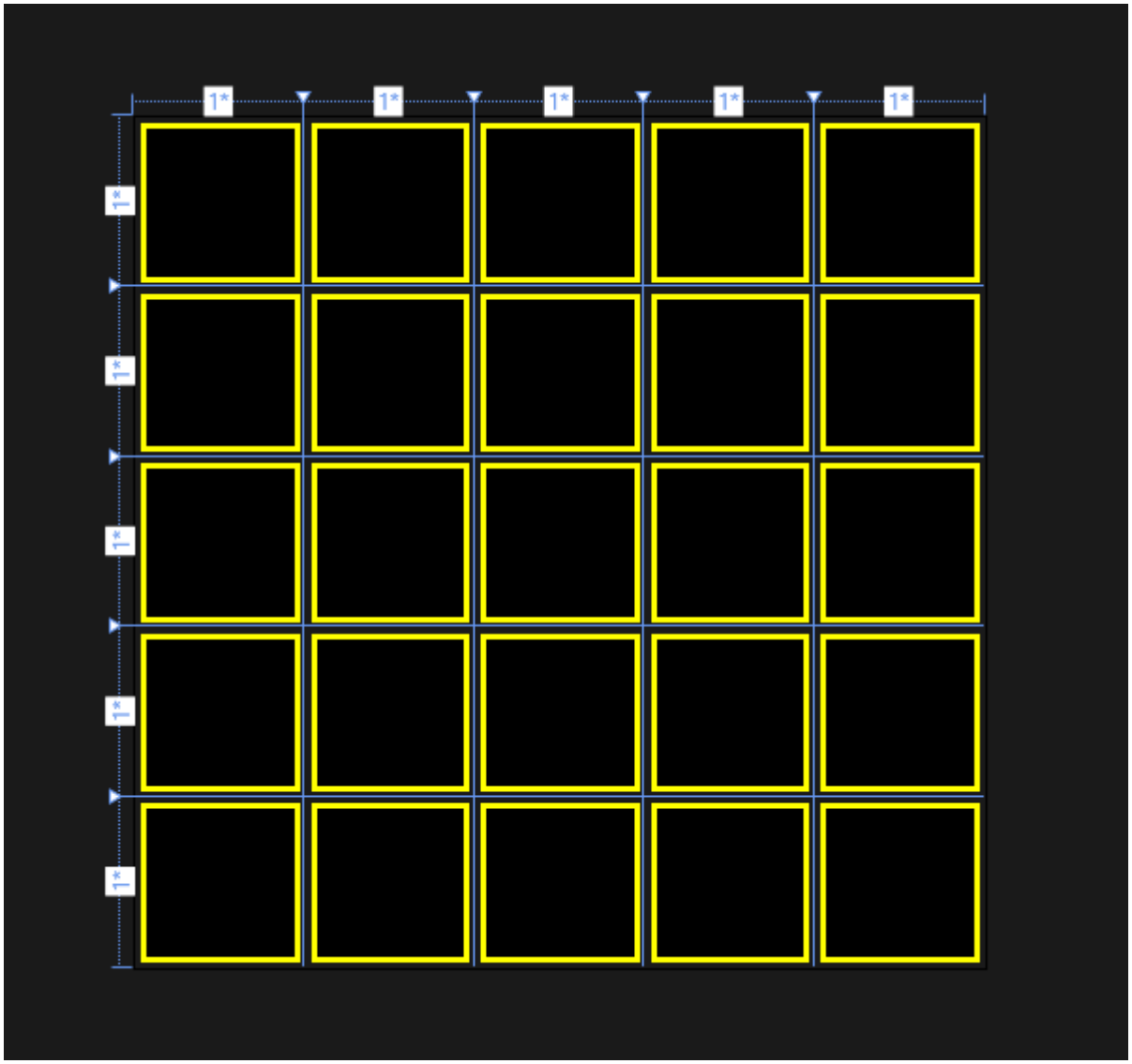

Рисунок 4.2. Сутність ігрового поля, як окремий елемент UserControl

У верхній або нижній частині головного вікна розміщені елементи керування грою: кнопка початку нової партії, кнопка завершення гри, а також, за потреби, індикатор поточного числа, яке має бути розміщене на полі. Кнопка

«Нова гра» ініціює виклик методу StartNewGame, який скидає стан поля, обнуляє або переобчислює рахунок, генерує початкову послідовність чисел і готує систему до нового ігрового циклу. Кнопка завершення гри може примусово зупиняти поточну партію та відображати фінальні результати, навіть якщо поле ще не повністю заповнене. Така організація інтерфейсу дає змогу користувачу чітко контролювати початок і кінець ігрового процесу.

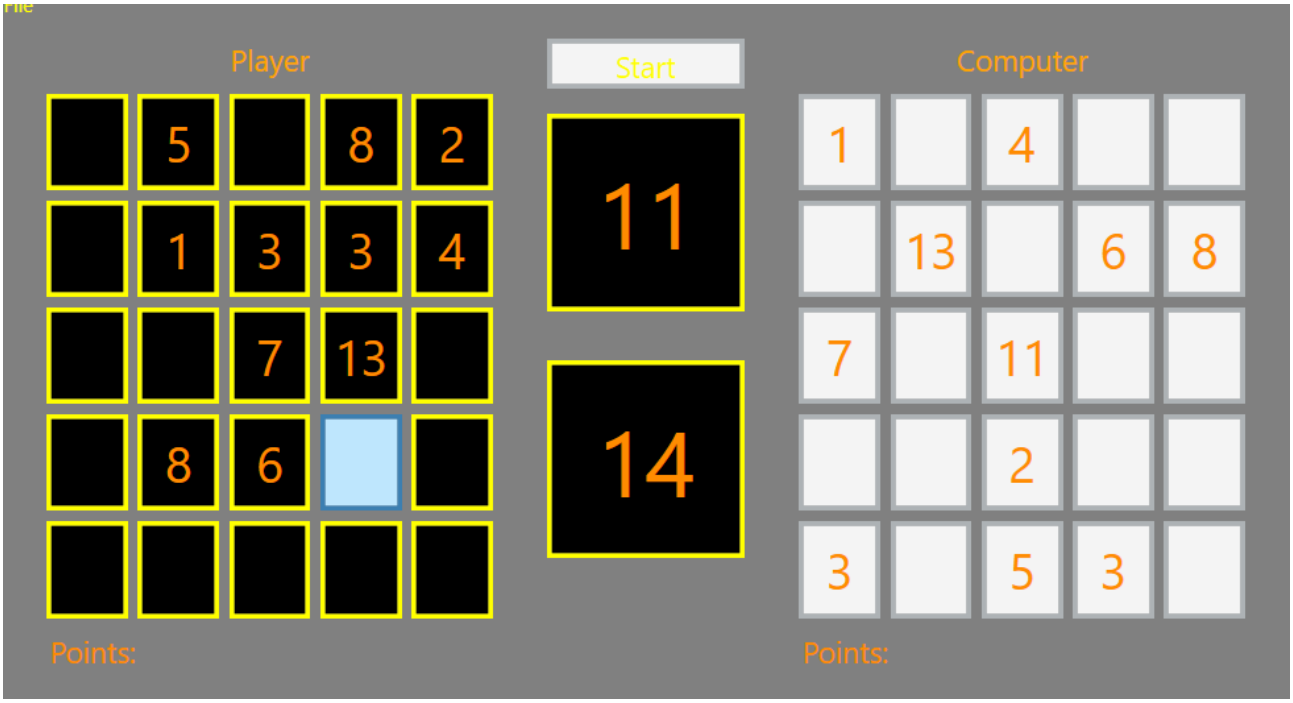

Рисунок 4.3. Процес гри з компютером

Особливості використання системи з точки зору користувача можна описати як послідовність дій у типовій ігровій сесії. Після запуску застосунку користувач бачить порожнє або ініціалізоване поле, нульовий (або стартовий) рахунок і індикатор того, чий хід виконується. Користувач здійснює хід, натискаючи на одну з вільних клітинок і розміщуючи задане число. Подія натискання надходить до класу MainWindow, де перевіряється допустимість ходу та, у разі успіху, здійснюється оновлення стану поля та рахунку. Одразу після цього система ініціює хід комп’ютера: алгоритм супротивника обирає координати, програма розміщує число на відповідній клітинці, а результат

знову відображається у вигляді оновлених числових значень рахунку.

Такий «діалог» між користувачем і комп’ютером, який відбувається в єдиному вікні, забезпечує наочний перебіг гри: усі зміни поля відбуваються в одному місці, без додаткових діалогів чи переходів між екранами. Завдяки цьому користувач може відразу оцінити наслідки своїх ходів, побачити реакцію алгоритму супротивника та простежити динаміку рахунку. З точки зору навчального використання це дає змогу не лише грати, а й аналізувати логіку гри, розглядаючи послідовні конфігурації поля та відповідні їм значення очок.

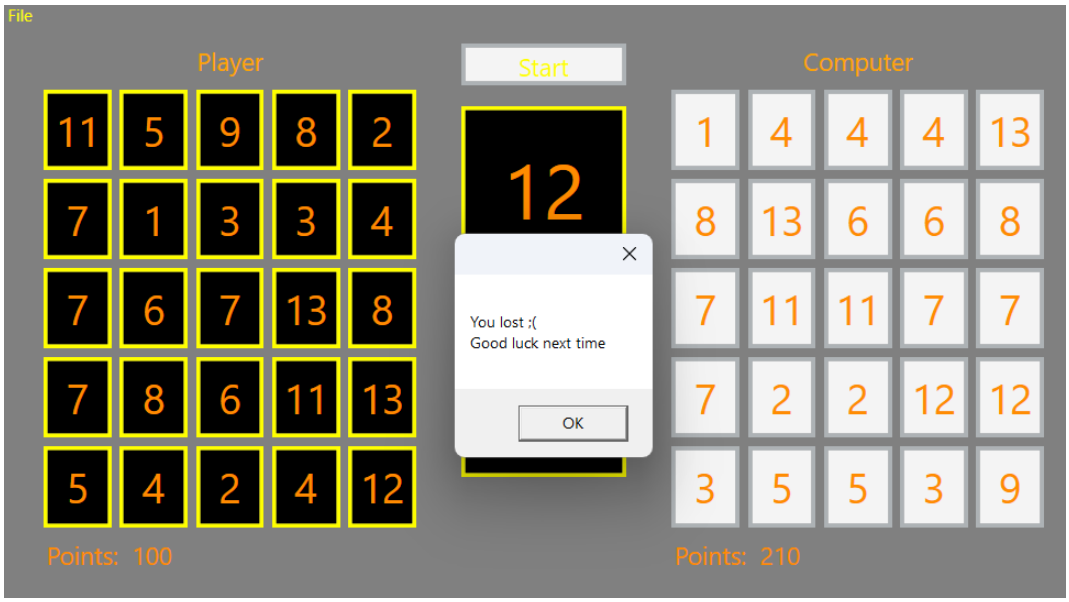

Рисунок 4.4. Результати гри і підрахунок очок

Окремо варто підкреслити, що інтерфейс не містить надмірних анімацій чи графічних ефектів, які могли б ускладнювати сприйняття математичної структури гри. Дизайн є стриманим, орієнтованим на чітке відображення числових значень і структур поля, а також на зручність роботи у навчальному середовищі (лекційна аудиторія, комп’ютерна лабораторія). За потреби інтерфейс може бути розширений додатковими елементами, наприклад, текстовою областю з поясненням правил гри, кнопками для перемикання режимів складності або підсвічуванням ліній, що принесли найбільше очок у поточному ході.

Суттєвою особливістю є також те, що інтерфейс організовано з

урахуванням можливості подальшої модифікації та використання в навчальних цілях для демонстрації принципів побудови WPF-застосунків. Елементи керування згруповані логічно, прив’язки даних і обробники подій реалізовані у прозорий спосіб, а структура XAML-розмітки відповідає базовим рекомендаціям щодо розділення представлення та логіки. Це дозволяє використовувати програмний продукт не лише як готову гру, а й як навчальний приклад для студентів спеціальності «Комп’ютерні науки», які можуть аналізувати та модифікувати інтерфейс, не порушуючи при цьому основної логіки гри.

Таким чином, інтерфейс користувача гри «Математико» та особливості її використання визначаються поєднанням простоти й наочності візуального представлення з чіткою інтеграцією з алгоритмічною частиною програми. Застосунок забезпечує інтуїтивно зрозумілу взаємодію для кінцевого користувача та одночасно створює умови для поглибленого аналізу структури програмного продукту в контексті професійної підготовки фахівців з комп’ютерних наук.

ВИСНОВКИ

У межах магістерської роботи було реалізовано десктопний застосунок, що поєднує формально задану логіко-математичну модель гри «Математико», алгоритми комп’ютера-суперника та графічний інтерфейс на основі технології WPF. Запропонований програмний продукт орієнтований на моделювання дискретної логіко-математичної гри з прозорими правилами, відкритою структурою коду та можливістю використання в навчальному процесі.

У ході роботи:

– проаналізовано сучасні програмні засоби та ігрові платформи з математики й логіки, виявлено їх типові архітектурні та дидактичні особливості, а також характерні недоліки з точки зору відкритості математичної моделі та доступності внутрішніх алгоритмів;

– обґрунтовано доцільність створення окремої програмної реалізації гри «Математико» як настільного застосунку з відкритою структурою коду, орієнтованою на подальший аналіз та модифікацію;

– формалізовано математичну модель гри у вигляді заповнення матриці розміру 5×5 елементами з фіксованого мультисета значень та цільової функції, яка обчислюється на основі результатів для рядків, стовпців і діагоналей;

– досліджено логіко-комбінаторні аспекти гри, описано типові структури (послідовності, пари, трійки, чотвірки, спеціальні комбінації), що враховуються під час оцінювання конфігурацій ігрового поля;

– розроблено та реалізовано алгоритми комп’ютера-суперника, які здійснюють евристичний вибір ходу на основі перебору можливих позицій, локального оцінювання конфігурацій і взаємодії з модулем підрахунку очок;

– спроєктовано структуру програмної системи, побудовано UML-діаграми варіантів класів і послідовності, що відображають основні компоненти, їх зв’язки та сценарії взаємодії;

– обґрунтовано вибір технологічного стеку (C#, .NET, WPF), визначено

переваги настільної реалізації для цілей аналізу архітектури й алгоритмів;

– реалізовано ігрове поле у вигляді користувацького елемента керування, механізм ходів гравця та комп’ютера, модуль підрахунку очок і візуалізацію результатів у графічному інтерфейсі;

– проведено апробацію програмного продукту, перевірено коректність підрахунку очок, роботи алгоритмів супротивника та відповідність реалізації сформульованій математичній моделі.

Розроблена система дозволяє користувачу:

– виконувати послідовні ходи, розміщуючи числа на полі 5×5 відповідно до правил гри «Математико»;

– спостерігати за автоматичною реакцією комп’ютера-суперника, який обирає ходи на основі реалізованих алгоритмів оцінювання конфігурацій;

– отримувати поточні та підсумкові результати у вигляді числового рахунку з урахуванням внеску рядків, стовпців і діагоналей;

– аналізувати вплив розташування чисел на полі на підсумкові очки, реконструюючи логіку цільової функції та стратегії комп’ютера.

Результати роботи відкривають перспективи для подальшого розвитку проєкту, зокрема:

– розширення набору стратегій комп’ютера-суперника (застосування багатокрокових алгоритмів пошуку, елементів мінімакс-підходу, врахування потенційних ходів користувача);

– запровадження різних рівнів складності гри шляхом зміни параметрів оцінювання, розміру поля, набору значень або правил підрахунку очок;

– розроблення мережевого або веб-варіанту гри з можливістю змагання між кількома користувачами та збиранням статистики ігрових сесій;

– інтеграції додаткових засобів візуалізації (підсвічування «вигідних» ліній, відображення історії ходів, аналітика за партіями);

– проведення педагогічних експериментів щодо використання гри «Математико» як інструмента для розвитку логічного мислення та

комбінаторного мислення студентів і школярів.

Таким чином, поставлену мету дослідження досягнуто: формалізовано математичну модель гри «Математико», спроєктовано та реалізовано настільний програмний продукт із чіткою архітектурою, алгоритмами комп’ютера-суперника та наочним інтерфейсом. Отриманий застосунок підтверджує ефективність поєднання логіко-комбінаторної моделі з сучасними засобами розробки програмного забезпечення для створення прозорих, аналізованих і розширюваних ігрових систем у галузі комп’ютерних наук.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Edin Kapic. Windows Forms Binding Improvements in .NET 7 for MVVM Support. Режим доступу: URL: https://www.infoq.com/news/2023/02/winforms-binding-mvvm-net-7/ - Назва з екрану.

2. Eileen Roche. Explaining XML. Режим доступу: URL: https://hbr.org/2000/07/explaining-xml - Назва з екрану.

3. Introduction to Visual Studio. Режим доступу: URL: https://www.geeksforgeeks.org/introduction-to-visual-studio/ - Назва з екрану.

4. Tutorial: Create a new WPF app with .NET. Режим доступу: URL: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/get-started/create-app-visual-studio?view=netdesktop-8.0 - Назва з екрану.

5. Vijay Kanade. What Is XML (Extensible Markup Language)? Meaning, Elements, and Benefits. Режим доступу: URL: https://www.spiceworks.com/tech/tech-general/articles/what-is-xml/ - Назва з екрану

6. What is a XAML file? Режим доступу: URL: https://docs.fileformat.com/web/xaml/ - Назва з екрану

7. WPF vs WinForms – Making the Right Decision in 2024. Режим доступу: URL: https://blog.ndepend.com/wpf-vs-winforms-choosing-the-proper-framework-for-your-project/ - Назва з екрану.

8. Євгенія Стенцель. Що таке C#? Кому підходить програмування на Сі Шарп? Режим доступу: URL: https://beetroot.academy/blog/shcho-take-c-chi-pidhodit-meni-cya-mova-programuvannya-chomu-vona-kruta - Назва з екрану.

9. КОШОВА О. РОЗРОБКА НАВЧАЛЬНОГО АНДРОЇД-ЗАСТОСУНКУ З ТЕМИ «СОРТУВАННЯ ВСТАВКАМИ» ДИСТАНЦІЙНОГО НАВЧАЛЬНОГО КУРСУ «АЛГОРИТМИ І СТРУКТУРИ ДАНИХ» / О. КОШОВА, О. ЧЕРНЕНКО, О. ОРІХІВСЬКА, В. ТУР, О. ЯНКО // Інформаційні

технології та суспільство, №5 (11), 2024. С. 34-42. Режим доступу: URL: https://doi.org/10.32689/maup.it.2023.5.5 - Назва з екрану

10. Ольховська, О. В., Кошова, О. П., Оборожний А.В., Жуля А. Розробка мобільного додатку для управління та дистрибуції новин на основі мікросервісної архітектури Вісник Кременчуцького національного університету імені Михайла Остроградського. Кременчук: КрНУ, Випуск 1/2025 (150), С. 160-168. DOI <https://doi.org/10.32782/1995-0519.2025.1.21> <https://visnikkrnu.kdu.edu.ua/visnik.php?id_nom=72>

11. Хохлов Д. та ін. XAML - мова розмітки інтерфейсу Silverlight додатків. Режим доступу: URL: http://www.znannya.org/?view=silverlight-intro1- Назва з екрану.

12. Черненко О. О. Методичні рекомендації щодо виконання кваліфікаційної роботи студентів спеціальності 122 Комп'ютерні науки освітня програма «Комп'ютерні науки» ступеня магістра / С. В. Гаркуша, О. В. Ольховська, О. О. Черненко. – Полтава : ПУЕТ, 2023. – 68 с. Режим доступу: URL: http://elib.puet.edu.ua/action.php?kt_path_info=ktcore.SecViewPlugin.actions.document&fDocumentId=824868 - Назва з екрану

13. Як вивчити мову програмування C# та стати .NET розробником. Режим доступу: URL: https://edu.cbsystematics.com/ua/blog/learn-csharp-become-dnet - Назва з екрану.

14. Deterding, S., Dixon, D., Khaled, R., Nacke, L. From Game Design Elements to Gamefulness: Defining "Gamification" // Proceedings of the 15th International Academic MindTrek Conference. – 2011. – P. 9–15. – Режим доступу: URL: https://dl.acm.org/doi/10.1145/2181037.2181040

15. Prodigy Math Game – Fun Math Learning for Kids. Режим доступу: URL: https://www.prodigygame.com/

16. DragonBox Algebra 5+ – Learn Algebra Through Play. Режим доступу: URL: https://dragonbox.com/products/algebra-5/

17. DragonBox Algebra 12+ – Learn Algebra Concepts Visually. Режим доступу: URL: https://dragonbox.com/products/algebra-12/

18. Kahoot! DragonBox – Math Apps for Learning Through Play. Режим доступу: URL: https://kahoot.com/dragonbox/

19. Matific – Math Learning Games for Kids. Режим доступу: URL: https://www.matific.com/

20. Math. Khan Academy. Режим доступу: URL: https://www.khanacademy.org/math

21. Brilliant | Learn Math and Science Through Problem Solving. Режим доступу: URL: https://brilliant.org/

22. Learning.ua – Освітня онлайн-платформа для дітей. Режим доступу: URL: https://learning.ua/

23. Алаба – освітні ігри, конкурси та олімпіади. Режим доступу: URL: https://alaba.op.ua/

24. Логічні ігри онлайн – Formula.co.ua. Режим доступу: URL: https://formula.co.ua/logic-games

ДОДАТОК А
Код програми

WpfMathematicoGameApp.csproj

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net6.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWPF>true</UseWPF>
  </PropertyGroup>

</Project>
```

App.xaml

```
<Application x:Class="WpfMathematicoGameApp.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WpfMathematicoGameApp"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

App.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
```

```
using System.Threading.Tasks;
using System.Windows;

namespace WpfMathematicoGameApp
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
    }
}
```

MainWindow.xaml

```
<Window x:Class="WpfMathematicoGameApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfMathematicoGameApp"
        xmlns:uc="clr-namespace:WpfMathematicoGameApp.UserControls"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800" Background="Gray"
        WindowStyle="None"
        BorderBrush="DarkOrange"   BorderThickness="3"   WindowStartupLocation="CenterScreen"
ResizeMode="NoResize">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="271*"/>
            <ColumnDefinition Width="126*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="30px"></RowDefinition>
            <RowDefinition></RowDefinition>
```

```
        <RowDefinition Height="30px"></RowDefinition>
    </Grid.RowDefinitions>
    <Menu Grid.Row="0" Background="Gray" Foreground="Yellow" Grid.ColumnSpan="2">
        <MenuItem Header="File">
            <MenuItem Header="Exit" Background="Gray"
Click="MenuItem_Click"></MenuItem>
        </MenuItem>
    </Menu>
    <Grid Grid.Row="1" Grid.ColumnSpan="2">
        <Grid.RowDefinitions>
            <RowDefinition Height="30"></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="30"></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition Width="30"></ColumnDefinition>
            <ColumnDefinition Width="120"></ColumnDefinition>
            <ColumnDefinition Width="30"></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition Width="30"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Label Content="Player" Grid.Column="1" Foreground="Orange" FontSize="18"
            HorizontalContentAlignment="Center" Padding="-5"></Label>
        <Label Content="Computer" Grid.Column="5" Foreground="Orange" FontSize="18"
            HorizontalContentAlignment="Center" Padding="-5"></Label>

        <uc:GameCardControl Grid.Column="1" Grid.Row="1"
x:Name="GameCardControl_Player" IsEnabled="False"/>
        <uc:GameCardControl Grid.Column="5" Grid.Row="1" IsEnabled="False"
x:Name="GameCardControl_Computer"/>
        <Button Grid.Row="0" Grid.Column="3" Foreground="Yellow"
            Background="Black" FontSize="18" BorderBrush="DarkOrange"
```

```
            BorderThickness="3"                    x:Name="ButtonStartGame"
Click="ButtonStartGame_Click">Start</Button>
        <StackPanel Grid.Row="1" Grid.Column="3">
          <Label Background="Black" MinHeight="120"
              Margin="0,15,0,0"
              BorderBrush="Yellow"
              BorderThickness="3"
              Foreground="DarkOrange"
              VerticalContentAlignment="Center"
              HorizontalContentAlignment="Center"
              FontSize="56"
              x:Name="LabelCurrentCard"></Label>
          <Label Background="Black" MinHeight="120"
              Margin="0,30,0,0"
              BorderBrush="Yellow"
              BorderThickness="3"
              Foreground="DarkOrange"
              VerticalContentAlignment="Center"
              HorizontalContentAlignment="Center"
              FontSize="56"
              x:Name="LabelCardsLeft">25</Label>
        </StackPanel>
        <StackPanel Grid.Row="2" Grid.Column="1" Orientation="Horizontal">
          <Label Foreground="DarkOrange" FontSize="18">Points:</Label>
          <Label              Foreground="DarkOrange"              FontSize="18"
x:Name="LabelPlayerPoints"></Label>
        </StackPanel>
        <StackPanel Grid.Row="2" Grid.Column="5" Orientation="Horizontal">
          <Label Foreground="DarkOrange" FontSize="18">Points:</Label>
          <Label              Foreground="DarkOrange"              FontSize="18"
x:Name="LabelComputerPoints"></Label>
        </StackPanel>
      </Grid>
    </Grid>
```

```
</Window>

MainWindow.xaml.cs
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using WpfMathematicoGameApp.Services;

namespace WpfMathematicoGameApp
{
   public partial class MainWindow : Window
   {
      private List<int> _nums = new List<int>();
      private int _lowerNum = 1;
      private int _upperNum = 13;
      private int _repeats = 4;
      private int _cardsLeft = 25;
      private int _curIndex;
      private int _curNum;
      private Random _rnd = new Random();

      public MatematicoComputer computer = new MatematicoComputer();
      public MainWindow()
      {
         InitializeComponent();
         GameCardControl_Player.Card_Click += CardClick;
         computer.Card = GameCardControl_Computer;
      }

      public void CardClick(object sender, RoutedEventArgs e)
      {
         if (((Button)sender).Content != null)
         {
```

```
            return;
        }
        ((Button)sender).Content = _curNum;
        _cardsLeft--;
        _updateCards();
        //Computer move
        computer.MakeMove(_curNum);
        //Check winner
        if (GameCardControl_Computer.Finished() && GameCardControl_Player.Finished())
        {
            //count points
            int playerScore = ScoreCalculator.Calculate(GameCardControl_Player);
            int computerScore = ScoreCalculator.Calculate(GameCardControl_Computer);
            LabelPlayerPoints.Content = playerScore.ToString();
            LabelComputerPoints.Content = computerScore.ToString();
            //show results
            if (playerScore > computerScore)
            {
                MessageBox.Show("You WON!!!\nCongratulations");
            }
            else if (playerScore < computerScore)
            {
                MessageBox.Show("You lost ;(\nGood luck next time");
            }
            else
            {
                MessageBox.Show("Draw");
            }
            ButtonStartGame.IsEnabled = true;
            GameCardControl_Player.IsEnabled = false;
            return;
        }

    }
```

```
private void ButtonStartGame_Click(object sender, RoutedEventArgs e)
{
   GameCardControl_Computer.Clear();
   GameCardControl_Player.Clear();
   _nums = new List<int>();
   for (int i = _lowerNum; i <= _upperNum; i++)
   {
      for (int j = 0; j < _repeats; j++)
      {
         _nums.Add(i);
      }
   }
   _cardsLeft = 25;
   GameCardControl_Player.IsEnabled = true;
   ButtonStartGame.IsEnabled = false;
   LabelPlayerPoints.Content = "";
   LabelComputerPoints.Content = "";
   _updateCards();
}

private void _updateCards()
{
   /*if (_cardsLeft <= 0)
   {
      LabelCurrentCard.Content = "";
      return;
   }*/
   LabelCardsLeft.Content = _cardsLeft.ToString();
   _curIndex = _rnd.Next(_nums.Count);
   _curNum = _nums[_curIndex];
   LabelCurrentCard.Content = _curNum;
   _nums.RemoveAt(_curIndex);
}
```

```
        private void MenuItem_Click(object sender, RoutedEventArgs e)
        {
            this.Close();
        }
    }
}
```

GameCardControl.xaml

```
<UserControl x:Class="WpfMathematicoGameApp.UserControls.GameCardControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:WpfMathematicoGameApp.UserControls"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="450">
    <Grid x:Name="Grid_GameCard">
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Button Grid.Column="0" Grid.Row="0" FontSize="30" Background="Black"
```

```
        Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
    <Button Grid.Column="1" Grid.Row="0" FontSize="30" Background="Black"
        Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
    <Button Grid.Column="2" Grid.Row="0" FontSize="30" Background="Black"
        Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
    <Button Grid.Column="3" Grid.Row="0" FontSize="30" Background="Black"
        Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
    <Button Grid.Column="4" Grid.Row="0" FontSize="30" Background="Black"
        Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>

    <Button Grid.Column="0" Grid.Row="1" FontSize="30" Background="Black"
        Foreground="DarkOrange"   BorderBrush="Yellow"   BorderThickness="3"   Margin="3"
></Button>
    <Button Grid.Column="1" Grid.Row="1" FontSize="30" Background="Black"
        Foreground="DarkOrange"   BorderBrush="Yellow"   BorderThickness="3"   Margin="3"
></Button>
    <Button Grid.Column="2" Grid.Row="1" FontSize="30" Background="Black"
        Foreground="DarkOrange"   BorderBrush="Yellow"   BorderThickness="3"   Margin="3"
></Button>
    <Button Grid.Column="3" Grid.Row="1" FontSize="30" Background="Black"
        Foreground="DarkOrange"   BorderBrush="Yellow"   BorderThickness="3"   Margin="3"
></Button>
    <Button Grid.Column="4" Grid.Row="1" FontSize="30" Background="Black"
        Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>

    <Button Grid.Column="0" Grid.Row="2" FontSize="30" Background="Black"
        Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
```

```
        <Button Grid.Column="1" Grid.Row="2" FontSize="30" Background="Black"
          Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
        <Button Grid.Column="2" Grid.Row="2" FontSize="30" Background="Black"
          Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
        <Button Grid.Column="3" Grid.Row="2" FontSize="30" Background="Black"
          Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
        <Button Grid.Column="4" Grid.Row="2" FontSize="30" Background="Black"
          Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>

        <Button Grid.Column="0" Grid.Row="3" FontSize="30" Background="Black"
          Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
        <Button Grid.Column="1" Grid.Row="3" FontSize="30" Background="Black"
          Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
        <Button Grid.Column="2" Grid.Row="3" FontSize="30" Background="Black"
          Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
        <Button Grid.Column="3" Grid.Row="3" FontSize="30" Background="Black"
          Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
        <Button Grid.Column="4" Grid.Row="3" FontSize="30" Background="Black"
          Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>

        <Button Grid.Column="0" Grid.Row="4" FontSize="30" Background="Black"
          Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
        <Button Grid.Column="1" Grid.Row="4" FontSize="30" Background="Black"
```

```
        Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
      <Button Grid.Column="2" Grid.Row="4" FontSize="30" Background="Black"
        Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
      <Button Grid.Column="3" Grid.Row="4" FontSize="30" Background="Black"
        Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
      <Button Grid.Column="4" Grid.Row="4" FontSize="30" Background="Black"
        Foreground="DarkOrange"          BorderBrush="Yellow"          BorderThickness="3"
Margin="3"></Button>
   </Grid>
</UserControl>
```

GameCardControl.xaml.cs

```
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;

namespace WpfMathematicoGameApp.UserControls
{
   /// <summary>
   /// Interaction logic for GameCardControl.xaml
   /// </summary>
   public partial class GameCardControl : UserControl
   {
      public readonly List<List<Button>> _buttons;
      public GameCardControl()
      {
         InitializeComponent();

         _buttons = new List<List<Button>>();

         for (int r = 0; r < 5; r++)
```

```
        {
            _buttons.Add(new List<Button>());
            for (int c = 0; c < 5; c++)
            {
                _buttons[r].Add((Button)Grid_GameCard.Children[r * 5 + c]);
                ((Button)Grid_GameCard.Children[r * 5 + c]).Click += GameCardControlButton_Click;
            }
        }
    }

    public void Clear()
    {
        foreach (var list in _buttons)
        {
            foreach (var button in list)
            {
                button.Content = null;
            }
        }
    }

    private void GameCardControlButton_Click(object sender, RoutedEventArgs e)
    {
        Card_Click?.Invoke(sender, e);
    }

    public delegate void GameCardButton_Clicked(object sender, RoutedEventArgs e);

    public event GameCardButton_Clicked Card_Click;

    public List<List<Button>> Buttons { get { return _buttons; } }

    public bool Finished()
```

```
        {
            bool isFinish = true;
            foreach (var list in _buttons)
            {
                foreach (var button in list)
                {
                    if (button.Content == null)
                    {
                        isFinish = false;
                    }
                }
            }
            return isFinish;
        }
    }
}
```

MatematicoComputer.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Controls;
using WpfMathematicoGameApp.UserControls;
using static System.Formats.Asn1.AsnWriter;

namespace WpfMathematicoGameApp.Services
{
    public class MatematicoComputer
    {
        public GameCardControl Card { get; set; }

        public void MakeMove(int currentNum)
        {
```

```
    if (!TryMakeDiagonalMove(currentNum))
    {
        if (!TryMakeRowColMove(currentNum))
        {
            MakeRandomMove(currentNum);
        }
    }
}

private bool TrySetCell(Button button, int value)
{
    if (button.Content == null)
    {
        button.Content = value.ToString();
        return true;
    }
    return false;
}

private bool TryMakeDiagonalMove(int currentNum)
{
    var buttons = Card.Buttons;
    List<List<int>> values = new List<List<int>>();

    for (int i = 0; i < buttons.Count; i++)
    {
        values.Add(new List<int>());
        for (int j = 0; j < buttons[i].Count; j++)
        {
            if (buttons[i][j].Content == null)
                values[i].Add(-1);
            else
                values[i].Add(int.Parse(buttons[i][j].Content.ToString()));
        }
```

```
        }

        Dictionary<int, int> mainDiagon = new Dictionary<int, int>(), secondaryDiagon = new Dictionary<int, int>();
        for (int i = 0; i < values.Count(); i++)
        {
            if (!mainDiagon.ContainsKey(values[i][i]))
                mainDiagon.Add(values[i][i], 0);
            mainDiagon[values[i][i]]++;

            if (!secondaryDiagon.ContainsKey(values[i][values.Count() - 1 - i]))
                secondaryDiagon.Add(values[i][values.Count() - 1 - i], 0);
            secondaryDiagon[values[i][values.Count() - 1 - i]]++;
        }

        if (currentNum == 1 || currentNum == 10 || currentNum == 11 || currentNum == 12 || currentNum == 13)
        {
            if (!mainDiagon.ContainsKey(currentNum))
            {
                for (int i = 0; i < values.Count; i++)
                {
                    if (TrySetCell(buttons[i][i], currentNum))
                        return true;
                }
            }
            if (!secondaryDiagon.ContainsKey(currentNum))
            {
                for (int i = 0; i < values.Count; i++)
                {
                    if (TrySetCell(buttons[i][4 - i], currentNum))
                        return true;
                }
            }
```

```
    }
    return false;
}

private bool TryMakeRowColMove(int currentNum)
{
    var buttons = Card.Buttons;
    List<List<int>> values = new List<List<int>>();

    for (int i = 0; i < buttons.Count; i++)
    {
        values.Add(new List<int>());
        for (int j = 0; j < buttons[i].Count; j++)
        {
            if (buttons[i][j].Content == null)
                values[i].Add(-1);
            else
                values[i].Add(int.Parse(buttons[i][j].Content.ToString()));
        }
    }

    // проверка по строкам
    Dictionary<int, int> currentRow;
    for (int i = 0; i < values.Count(); i++)
    {
        currentRow = new Dictionary<int, int>();
        for (int j = 0; j < values[i].Count(); j++)
        {
            if (!currentRow.ContainsKey(values[i][j]))
                currentRow.Add(values[i][j], 0);
            currentRow[values[i][j]]++;
        }

        if (!currentRow.ContainsKey(-1)) continue;
```

```
            if (currentRow.ContainsKey(1) && currentNum == 1)
                return AddToRow(values, buttons, currentNum, i);

            if (currentRow.ContainsValue(3) && currentRow[-1] != 3 &&
                currentRow.ContainsKey(currentNum) && currentRow[currentNum] == 3)
                return AddToRow(values, buttons, currentNum, i);

            if ((currentRow.ContainsValue(3) && currentRow.ContainsValue(1) && currentRow[-1]
!= 3 &&
                 currentRow.ContainsKey(currentNum) && currentRow[currentNum] == 1) ||
                (currentRow.Keys.Where(val => val == 2).Count() == 2 && currentRow[-1] != 2 &&
                 currentRow.ContainsKey(currentNum) && currentRow[currentNum] == 2))
                return AddToRow(values, buttons, currentNum, i);

            if (currentRow.ContainsValue(2) && currentRow.ContainsKey(currentNum) &&
currentRow[currentNum] == 2)
                return AddToRow(values, buttons, currentNum, i);

            if (currentRow.ContainsKey(currentNum) && currentRow[currentNum] == 1)
                return AddToRow(values, buttons, currentNum, i);
        }

        // проверка по колонкам
        Dictionary<int, int> currentCol;
        for (int i = 0; i < values.Count(); i++)
        {
            currentCol = new Dictionary<int, int>();
            for (int j = 0; j < values[i].Count(); j++)
            {
                if (!currentCol.ContainsKey(values[j][i]))
                    currentCol.Add(values[j][i], 0);
                currentCol[values[j][i]]++;
            }
```

```
            if (!currentCol.ContainsKey(-1)) continue;

            if (currentCol.ContainsKey(1) && currentNum == 1)
                return AddToCol(values, buttons, currentNum, i);

            if (currentCol.ContainsValue(3) && currentCol[-1] != 3 &&
                currentCol.ContainsKey(currentNum) && currentCol[currentNum] == 3)
                return AddToCol(values, buttons, currentNum, i);

            if ((currentCol.ContainsValue(3) && currentCol.ContainsValue(1) && currentCol[-1] !=
3 &&
                 currentCol.ContainsKey(currentNum) && currentCol[currentNum] == 1) ||
                (currentCol.Keys.Where(val => val == 2).Count() == 2 && currentCol[-1] != 2 &&
                 currentCol.ContainsKey(currentNum) && currentCol[currentNum] == 2))
                return AddToCol(values, buttons, currentNum, i);

            if (currentCol.ContainsValue(2) && currentCol.ContainsKey(currentNum) &&
currentCol[currentNum] == 2)
                return AddToCol(values, buttons, currentNum, i);

            if (currentCol.ContainsKey(currentNum) && currentCol[currentNum] == 1)
                return AddToCol(values, buttons, currentNum, i);
        }
        return false;
    }

    private bool AddToRow(List<List<int>> values, List<List<Button>> buttons, int currentNum,
int curRow)
    {
        for (int iter = 0; iter < values[curRow].Count(); iter++)
        {
            if (TrySetCell(buttons[curRow][iter], currentNum))
                return true;
```

```
        }
        return false;
    }

    private bool AddToCol(List<List<int>> values, List<List<Button>> buttons, int currentNum, int curCol)
    {
        for (int iter = 0; iter < values.Count(); iter++)
        {
            if (TrySetCell(buttons[iter][curCol], currentNum))
                return true;
        }
        return false;
    }

    private bool OnlyDiagonalsFree()
    {
        for (int i = 0; i < Card.Buttons.Count; i++)
        {
            for (int j = 0; j < Card.Buttons[i].Count; j++)
            {
                if (i != j && j != 4 - i && Card.Buttons[i][j].Content == null)
                    return false;
            }
        }
        return true;
    }

    private void MakeRandomMove(int currentNum)
    {
        Random rnd = new Random();
        int i, j;

        if (!OnlyDiagonalsFree())
```

```
            {
                do
                {
                    i = rnd.Next(5);
                    j = rnd.Next(5);
                }
                while (Card.Buttons[i][j].Content != null);
            }
            else
            {
                do
                {
                    i = rnd.Next(5);
                    j = rnd.Next(5);
                }
                while (Card.Buttons[i][j].Content != null);
            }
            Card.Buttons[i][j].Content = currentNum.ToString();
        }
    }
}
```

ScoreCalculator.cs

```
using System.Collections.Generic;
using System.Linq;
using WpfMathematicoGameApp.UserControls;

namespace WpfMathematicoGameApp.Services
{
    public static class ScoreCalculator
    {
        public static int Calculate(GameCardControl card)
        {
```

```
        int score = 0;
        score += countDiagonals(card);
        score += countRows(card);
        score += countColumns(card);
        return score;
    }

    private static int countDiagonals(GameCardControl card)
    {
        //key - numner, value - repeats
        var buttons = card.Buttons;
        List<List<int>> values = new List<List<int>>();
        for (int i = 0; i < buttons.Count; i++)
        {
            values.Add(new List<int>());
            for (int j = 0; j < buttons[i].Count; j++)
            {
                values[i].Add(int.Parse(buttons[i][j].Content.ToString()));
            }
        }
        int score = 0;
        Dictionary<int, int> mainDiagon = new Dictionary<int, int>(), secondaryDiagon = new
Dictionary<int, int>();
        for (int i = 0; i < values.Count(); i++)
        {
            if (!mainDiagon.ContainsKey(values[i][i]))
            {
                mainDiagon.Add(values[i][i], 0);
            }
            mainDiagon[values[i][i]]++;

            if (!secondaryDiagon.ContainsKey(values[i][values.Count() - 1 - i]))
            {
                secondaryDiagon.Add(values[i][values.Count() - 1 - i], 0);
```

```
            }
            secondaryDiagon[values[i][values.Count() - 1 - i]]++;
        }
        score += countLineScore(mainDiagon, true);
        score += countLineScore(secondaryDiagon, true);
        return score;
    }

    private static int countLineScore(Dictionary<int, int> values, bool isDiagonal = false)
    {
        int score = isDiagonal ? 10 : 0;
        if (values.ContainsKey(1) && values[1] == 4)
        {
            score += 200;
        }
        else if (values.ContainsKey(1) && values.ContainsKey(10) && values.ContainsKey(11) && values.ContainsKey(12) && values.ContainsKey(13))
        {
            score += 150;
        }
        else if (values.ContainsKey(1) && values.ContainsKey(13) && values[1] == 3 && values[13] == 2)
        {
            score += 100;
        }
        else if (values.Keys.Count == 5 && values.Keys.Max() - values.Keys.Min() == 4)
        {
            score += 50;
        }
        else if (values.ContainsValue(4))
        {
            score += 160;
        }
        else if (values.Keys.Count == 2)
```

```
    {
        score += 80;
    }
    else if (values.ContainsValue(3))
    {
        score += 40;
    }
    else if (values.Keys.Count == 3)
    {
        score += 20;
    }
    else if (values.ContainsValue(2))
    {
        score += 10;
    }
    return score;
}

private static int countRows(GameCardControl card)
{
    //key - numner, value - repeats
    var buttons = card.Buttons;
    List<List<int>> values = new List<List<int>>();
    for (int i = 0; i < buttons.Count; i++)
    {
        values.Add(new List<int>());
        for (int j = 0; j < buttons[i].Count; j++)
        {
            values[i].Add(int.Parse(buttons[i][j].Content.ToString()));
        }
    }
    int score = 0;
    Dictionary<int, int> currentRow;
    for (int i = 0; i < values.Count(); i++)
```

```
    {
        currentRow = new Dictionary<int, int>();
        for (int j = 0; j < values[i].Count(); j++)
        {
            if (!currentRow.ContainsKey(values[i][j]))
            {
                currentRow.Add(values[i][j], 0);
            }
            currentRow[values[i][j]]++;
        }
        score += countLineScore(currentRow);
    }
    return score;
}

private static int countColumns(GameCardControl card)
{
    //key - numner, value - repeats
    var buttons = card.Buttons;
    List<List<int>> values = new List<List<int>>();
    for (int i = 0; i < buttons.Count; i++)
    {
        values.Add(new List<int>());
        for (int j = 0; j < buttons[i].Count; j++)
        {
            values[i].Add(int.Parse(buttons[i][j].Content.ToString()));
        }
    }
    int score = 0;
    Dictionary<int, int> currentCol;
    for (int i = 0; i < values.Count(); i++)
    {
        currentCol = new Dictionary<int, int>();
        for (int j = 0; j < values[i].Count(); j++)
```

```
            {
                if (!currentCol.ContainsKey(values[j][i]))
                {
                    currentCol.Add(values[j][i], 0);
                }
                currentCol[values[j][i]]++;
            }
            score += countLineScore(currentCol);
        }
        return score;
    }
  }
}
```