**Реалізація Use-Case в рамках MVC в LambdaChess**

**Виконали**

Бенько Володимир  
Бучко Роман  
Долуда Андрій  
Зозуля Олег

**Виправлення або відключення помилок аналізатора коду**

Проект LambdaChess на GitHub не містить явної конфігурації для аналізатора коду, але можна виділити кілька підходів до роботи з помилками аналізатора, які часто використовуються в .NET проектах:

1. **Файли EditorConfig**
   * Проект використовує автоматично згенерований файл .editorconfig (видно в шляхах \*.GeneratedMSBuildEditorConfig.editorconfig), який містить правила форматування і стилю коду для всього рішення
   * Це допомагає дотримуватися єдиного стилю коду в команді
2. **Правила, що можуть бути відключені**:
   * CA1822: Помітка методів як статичних - не завжди доречно, якщо метод може стати інстанційним у майбутньому
   * IDE0090: Використання спрощеного оператора new - може ускладнити читабельність для початківців
   * CS1591: Відсутні XML-коментарі - не всі проекти потребують документування кожного публічного елемента
3. **Правила, що варто залишити активними**:
   * CA2007: Не забувайте await Tasks
   * CA2016: Перенаправлення методів на їх завдання
   * IDE0005: Видалення невикористаних using-директив
   * CS8600-CS8618: Перевірки на null-referenсes
4. **Спеціальні директиви**
   * Використання #pragma warning disable {код} для тимчасового відключення попереджень у конкретних місцях
   * Атрибут [SuppressMessage] для специфічних ситуацій

**Написання контролерів (controller) для обробки запиту користувача, що відповідає use-case**

У проекті LambdaChess реалізовано кілька контролерів для обробки різних use-case, пов'язаних з шаховою грою:

1. **HomeController** (LambdaChess.Web.UI/Controllers/HomeController.cs)
   * Відповідає за базові дії веб-сайту:
     + Відображення головної сторінки (Index)
     + Відображення сторінки конфіденційності (Privacy)
     + Обробка помилок (Error)
2. **GameController** (LambdaChess.Web.UI/Controllers/GameController.cs)
   * Ключовий контролер для реалізації основного use-case - шахової гри:
     + Відображення ігрової дошки (Index) - центральний компонент гри
     + Створення нової гри (CreateGame) - ініціалізація ігрової сесії
     + Показ лобі з доступними іграми (Lobby) - для вибору партії
     + Приєднання до існуючої гри (JoinGame) - для участі у створеній партії
3. **BaseController** (LambdaChess.Web.UI/Controllers/BaseController.cs)
   * Базовий абстрактний контролер, від якого наслідуються інші:
     + Надає доступ до інформації про поточного користувача
     + Містить спільні методи та властивості для всіх контролерів
4. **LobbyController** (LambdaChess.Web.Controllers/LobbyController.cs)
   * Альтернативна реалізація функціоналу лобі (можливо, для використання в API)

**Особливості реалізації контролерів:**

* **Використання атрибутів авторизації** ([Authorize]) для забезпечення доступу тільки зареєстрованим користувачам
* **Dependency Injection** для впровадження залежностей через конструктор
* **Обробка помилок** через try-catch блоки з логуванням
* **Перевірка прав доступу** до ресурсів (чи має користувач право доступу до гри)
* **Використання результатів дій** (ActionResult) для повернення відповідних HTTP-відповідей:
  + View() для рендерингу представлень
  + RedirectToAction() для перенаправлення на інші дії
  + NotFound() та Forbid() для обробки помилок

**Написання класів сутностей та сервісів (model) для виконання бізнес-логіки**

**Класи сутностей (моделі даних)**

У проекті LambdaChess використовуються такі основні сутності:

1. **GameSession** (LambdaChess.DAL.Models/GameSession.cs)
   * Представляє шахову партію між двома гравцями
   * Ключові властивості:
     + Id - унікальний ідентифікатор гри
     + WhitePlayerId і BlackPlayerId - ідентифікатори гравців
     + WhitePlayer і BlackPlayer - навігаційні властивості
     + FEN - нотація Форсайта-Едвардса для представлення позиції на дошці
     + IsFinished - прапорець завершення гри
     + Winner - переможець (enum)
     + CreatedAt і FinishedAt - часові мітки
     + CurrentTurn - обчислювана властивість для визначення поточного гравця
2. **User** (LambdaChess.DAL.Models/User.cs)
   * Представляє користувача системи
   * Ключові властивості:
     + Id - унікальний ідентифікатор
     + UserName - ім'я користувача
     + Rating - рейтинг користувача в шахах
     + WhiteGames і BlackGames - зв'язки з іграми
3. **Winner** (LambdaChess.DAL.Models/Enums/Winner.cs)
   * Enum для зберігання результату гри:
     + White - перемога білих
     + Black - перемога чорних
     + Draw - нічия
4. **IModel** (LambdaChess.DAL.Models/Abstractions/IModel.cs)
   * Базовий інтерфейс для всіх моделей, що забезпечує наявність ідентифікатора

**Сервіси для бізнес-логіки**

1. **IGameService**
   * Інтерфейс з методами для управління шаховими партіями:
     + CreateGame - створення нової гри
     + JoinGame - приєднання до існуючої гри
     + MakeMove - виконання ходу
     + GetGameById - отримання гри за ідентифікатором
     + GetAvailableGames - отримання доступних ігор
     + GetUserGames - отримання ігор користувача
2. **GameService**
   * Реалізація IGameService з бізнес-логікою для шахових операцій
   * Використовує репозиторії для доступу до даних
   * Виконує валідацію дій користувача
   * Відповідає за стан гри та прогрес
   * Повертає результати операцій з використанням FluentResults
3. **WebApplicationBuilderExtensions** (LambdaChess.BLL.Services/Hosting/WebApplicationBuilderExtensions.cs)
   * Розширення для реєстрації сервісів у DI-контейнері
   * Дозволяє централізовано налаштувати всі залежності

**Дотримання правил написання "чистого" коду**

У проекті LambdaChess можна спостерігати такі принципи "чистого" коду:

1. **SOLID-принципи**
   * **Single Responsibility Principle (SRP)** - кожен клас має одну відповідальність:
     + Контролери відповідають за обробку HTTP-запитів
     + Сервіси містять бізнес-логіку
     + Репозиторії працюють з базою даних
   * **Open/Closed Principle (OCP)** - відкритість для розширення, закритість для модифікації:
     + Використання інтерфейсів дозволяє розширювати функціональність
     + Базові класи для наслідування
   * **Liskov Substitution Principle (LSP)** - належне використання наслідування
   * **Interface Segregation Principle (ISP)** - спеціалізовані інтерфейси
   * **Dependency Inversion Principle (DIP)** - залежність від абстракцій:
     + Ін'єкція репозиторіїв через інтерфейси
     + Ін'єкція сервісів через інтерфейси
2. **Шаблони проектування**
   * **Repository** - для доступу до даних
   * **Dependency Injection** - для управління залежностями
   * **Unit of Work** - для транзакцій з базою даних
3. **Загальні принципи**
   * **DRY (Don't Repeat Yourself)** - використання базових класів і методів
   * **KISS (Keep It Simple, Stupid)** - прості рішення для типових задач
   * **YAGNI (You Aren't Gonna Need It)** - відсутність надлишкового коду
   * **Змістовні імена** - описові назви класів, методів і змінних
   * **Організація коду** - логічне групування у проекти та простори імен

**Логування дій користувача та помилок**

Проект LambdaChess використовує вбудований механізм логування ASP.NET Core:

1. **Рівні логування**
   * LogInformation - для відстеження нормальних дій користувача
   * LogWarning - для неочікуваних, але не критичних ситуацій
   * LogError - для помилок та винятків
2. **Логування в контролерах**
   * Вхід користувача на сторінку
   * Створення нової гри
   * Приєднання до гри
   * Спроби неавторизованого доступу
3. **Логування в сервісах**
   * Бізнес-операції (створення гри, хід у грі)
   * Валідаційні помилки
   * Системні виключення
4. **Структуроване логування**
   * Використання параметризованих повідомлень
   * Включення контекстної інформації (ID користувача, ID гри)
   * Відстеження винятків з деталями
5. **Переваги реалізованого підходу**
   * Централізований моніторинг активності
   * Можливість аудиту дій користувачів
   * Швидке діагностування проблем
   * Аналіз патернів використання
   * Інформація для розслідування інцидентів

**Написання класів (репозиторіїв) для отримання чи зміни даних в базі даних**

Проект LambdaChess використовує шаблон Repository для абстрагування доступу до даних:

1. **Інтерфейси репозиторіїв** (LambdaChess.DAL.Repositories.Abstractions)
   * IRepository<T> - базовий інтерфейс з CRUD-операціями:
     + GetById - отримання за ідентифікатором
     + GetAll - отримання всіх записів
     + Add - додавання нового запису
     + Update - оновлення існуючого запису
     + Delete - видалення запису
     + SaveChanges - збереження змін
   * IGameSessionRepository - специфічний інтерфейс для шахових партій:
     + GetAvailableGames - отримання доступних ігор
     + GetUserGames - отримання ігор користувача
   * IUserRepository - специфічний інтерфейс для користувачів:
     + GetByUserName - пошук за ім'ям користувача
2. **Реалізації репозиторіїв** (LambdaChess.DAL.Repositories.Implementations)
   * BaseRepository<T> - базова реалізація для всіх типів:
     + Інкапсулює роботу з DbContext
     + Реалізує базові CRUD-операції
   * GameSessionRepository - реалізація для ігрових сесій:
     + Додаткові методи для отримання специфічних ігор
     + Включає пов'язані сутності (гравців) через Include
   * UserRepository - реалізація для користувачів:
     + Додатковий метод для пошуку за ім'ям
3. **ApplicationDbContext** (LambdaChess.DAL.Repositories.Implementations/Persistance)
   * Контекст Entity Framework Core для доступу до бази даних
   * Налаштування зв'язків між сутностями
   * Інтеграція з Identity для автентифікації
   * Визначення DbSet для основних сутностей:
     + GameSessions
     + ChessUsers
4. **Особливості реалізації**
   * **Entity Framework Core** як ORM-провайдер
   * **SQLite** як база даних для розробки (видно з підключених DLL)
   * **Lazy Loading** для оптимізації запитів
   * **Eager Loading** через Include для завантаження пов'язаних даних
   * **Міграції** для еволюції схеми бази даних (LambdaChess.Web.UI/Migrations)
5. **Переваги реалізованого підходу**
   * **Абстракція від конкретної СУБД** - можна легко змінити провайдер
   * **Тестованість** - можливість мокати репозиторії для unit-тестів
   * **Повторне використання коду** - загальні операції в базовому класі
   * **Бізнес-логіка не залежить від деталей доступу до даних**
   * **Краща керованість транзакцій** через SaveChanges

**Створення шаблонів відображень (view) для виконання use-case користувачем з використанням Razor view engine у ASP .NET Core MVC**

Проект LambdaChess містить набір Razor-шаблонів для різних екранів додатку:

1. **Основні представлення для гри**
   * Views/Game/Index.cshtml - ключовий view для відображення шахової партії:
     + Інтерактивна шахова дошка
     + Інформація про гравців
     + Стан гри (чий хід, завершена чи ні)
     + Історія ходів
     + Чат для спілкування
   * Views/Game/Lobby.cshtml - сторінка лобі для пошуку ігор:
     + Список доступних партій
     + Кнопка створення нової гри
     + Інформація про гравців та їх рейтинг
2. **Загальні представлення**
   * Views/Home/Index.cshtml - головна сторінка сайту
   * Views/Home/Privacy.cshtml - сторінка політики конфіденційності
   * Views/Shared/Error.cshtml - шаблон для відображення помилок
3. **Шаблони компонування**
   * Views/Shared/\_Layout.cshtml - основний макет сайту:
     + Навігаційна панель
     + Підключення загальних скриптів і стилів
     + Футер
   * Views/Shared/\_LoginPartial.cshtml - частковий шаблон для авторизації
   * Views/\_ViewImports.cshtml - імпорт просторів імен і тегів
   * Views/\_ViewStart.cshtml - спільні налаштування для всіх представлень
4. **Клієнтські ресурси**
   * **JavaScript файли**:
     + wwwroot/js/game.js - основна логіка гри на клієнті
     + wwwroot/js/site.js - загальні функції для сайту
   * **Зовнішні бібліотеки**:
     + chessboardjs - для візуалізації шахової дошки
     + chess.js - для правил гри і валідації ходів
     + signalr - для комунікації в реальному часі
     + bootstrap - для стилізації інтерфейсу
     + jquery і jquery-validation - для клієнтської логіки
   * **CSS стилі**:
     + wwwroot/css/site.css - власні стилі сайту
     + Views/Shared/\_Layout.cshtml.css - стилі для макету
   * **Зображення**:
     + Набори шахових фігур у різних стилях (alpha, uscf, wikipedia)
5. **Особливості реалізації представлень**
   * **Типізовані моделі** для статичної перевірки типів
   * **Часткові представлення** для повторно використовуваних компонентів
   * **Tag Helpers** для спрощення генерації HTML
   * **Умовна логіка** для відображення різного контенту
   * **Інтеграція з SignalR** для оновлень у реальному часі
   * **Секції** для включення специфічних скриптів і стилів
6. **Реалізація реального часу через SignalR**
   * Hubs/GameHub.cs - хаб для комунікації в реальному часі:
     + Передача ходів між гравцями
     + Оновлення стану дошки
     + Чат між гравцями
     + Сповіщення про приєднання/від'єднання гравців

**Написання unit-тестів з використанням xUnit або NUnit**

У наданій структурі проекту LambdaChess немає окремого проекту для unit-тестів. Однак, для повноцінної MVC реалізації важливо мати тестове покриття, особливо для бізнес-логіки.

**Рекомендований підхід до створення тестового проекту**

1. **Структура тестового проекту**
   * Створення проекту LambdaChess.Tests з використанням xUnit або NUnit
   * Організація тестів за структурою основного проекту:
     + Controllers/ - тести для контролерів
     + Services/ - тести для сервісів
     + Repositories/ - тести для репозиторіїв
     + Models/ - тести для бізнес-моделей
2. **Інструменти для мокування**
   * Використання Moq або NSubstitute для створення моків
   * Налаштування TestFixtures для спільного коду
   * In-memory бази даних для тестування репозиторіїв

**Покриття тестами програмного методу, що відповідає use-case**

Для шахової гри основними методами, що потребують тестування, є:

1. **Тестування GameService.CreateGame()**
   * Перевірка успішного створення гри
   * Перевірка повернення помилки для неіснуючого користувача
   * Перевірка обробки винятків при проблемах з БД
2. **Тестування GameService.JoinGame()**
   * Перевірка успішного приєднання до гри
   * Перевірка помилок при:
     + Неіснуючій грі
     + Вже заповненій грі
     + Спробі приєднатися до власної гри
     + Спробі приєднатися до завершеної гри
3. **Тестування GameService.MakeMove()**
   * Перевірка успішного виконання ходу
   * Перевірка помилок при:
     + Невалідному ході
     + Ході не в свою чергу
     + Ході в незапочатій грі
     + Ході в завершеній грі
   * Перевірка правильності зміни стану гри після ходу
   * Перевірка коректного визначення результату гри (мат, пат)
4. **Тестування GameController**
   * Перевірка повернення правильних ViewResult/RedirectResult
   * Перевірка правильного передання моделей у представлення
   * Перевірка авторизації та прав доступу

**Підбір параметрів методу для різних сценаріїв виконання**

Для повного покриття тестами потрібно враховувати:

1. **Позитивні сценарії**
   * Успішне виконання операцій з валідними даними
   * Граничні випадки, що залишаються валідними
2. **Негативні сценарії**
   * Невалідні вхідні дані:
     + null значення
     + Пусті рядки
     + Неіснуючі ідентифікатори
   * Порушення бізнес-правил:
     + Невалідні ходи
     + Спроби виконати дію у неправильному стані гри
   * Проблеми інфраструктури:
     + Виключення при роботі з БД
     + Проблеми мережі для SignalR
3. **Специфічні шахові сценарії**
   * Перевірка спеціальних ходів:
     + Рокіровка
     + Взяття на проході
     + Перетворення пішака
   * Перевірка визначення шаху
   * Перевірка визначення мату
   * Перевірка визначення пату
4. **Приклади тестових випадків**

Для методу JoinGame:

// Позитивний сценарій

[Fact]

public void JoinGame\_ValidParameters\_ReturnsSuccess()

{

// Arrange

var gameId = "game123";

var userId = "user456";

var game = new GameSession { Id = gameId, BlackPlayerId = null };

\_mockGameRepository.Setup(r => r.GetById(gameId)).Returns(game);

\_mockUserRepository.Setup(r => r.GetById(userId)).Returns(new User { Id = userId });

// Act

var result = \_gameService.JoinGame(gameId, userId);

// Assert

Assert.True(result.IsSuccess);

}

// Негативний сценарій - гра не існує

[Fact]

public void JoinGame\_GameNotFound\_ReturnsFailure()

{

// Arrange

var gameId = "nonexistent";

var userId = "user456";

\_mockGameRepository.Setup(r => r.GetById(gameId)).Returns((GameSession)null);

// Act

var result = \_gameService.JoinGame(gameId, userId);

// Assert

Assert.False(result.IsSuccess);

Assert.Contains(result.Errors, e => e.Message.Contains("not found"));

}

**Підхід до тестування з використанням моків**

1. **Налаштування загальних залежностей**
2. private readonly Mock<IGameSessionRepository> \_mockGameRepository;
3. private readonly Mock<IUserRepository> \_mockUserRepository;
4. private readonly Mock<ILogger<GameService>> \_mockLogger;
5. private readonly GameService \_gameService;
6. public GameServiceTests()
7. {
8. \_mockGameRepository = new Mock<IGameSessionRepository>();
9. \_mockUserRepository = new Mock<IUserRepository>();
10. \_mockLogger = new Mock<ILogger<GameService>>();
12. \_gameService = new GameService(
13. \_mockGameRepository.Object,
14. \_mockUserRepository.Object,
15. \_mockLogger.Object);
16. }
17. **Налаштування поведінки моків**
18. \_mockGameRepository.Setup(r => r.GetById(It.IsAny<string>()))
19. .Returns<string>(id => id == "valid-id" ? new GameSession { Id = id } : null);
20. **Перевірка взаємодії з моками**
21. \_mockGameRepository.Verify(r => r.Update(It.IsAny<GameSession>()), Times.Once);
22. \_mockGameRepository.Verify(r => r.SaveChanges(), Times.Once);
23. **Тестування контролерів з моками сервісів**
24. [Fact]
25. public void Index\_GameExists\_ReturnsViewWithGame()
26. {
27. // Arrange
28. var gameId = "game123";
29. var game = new GameSession { Id = gameId };
30. \_mockGameService.Setup(s => s.GetGameById(gameId)).Returns(game);
32. // Act
33. var result = \_controller.Index(gameId);
35. // Assert
36. var viewResult = Assert.IsType<ViewResult>(result);
37. var model = Assert.IsType<GameSession>(viewResult.Model);
38. Assert.Equal(gameId, model.Id);
39. }

**Переваги впровадження тестування**

Додавання unit-тестів до проекту LambdaChess забезпечило б:

1. **Підвищення якості коду**
   * Раннє виявлення дефектів
   * Підтвердження правильності функціонування
   * Уникнення регресії при змінах
2. **Документування очікуваної поведінки**
   * Тести як "жива" документація коду
   * Чітке розуміння вимог до методів
3. **Безпечний рефакторинг**
   * Впевненість, що зміни не порушили існуючу функціональність
   * Швидкий зворотний зв'язок про проблеми
4. **Підтримка CI/CD**
   * Автоматичне виконання тестів у pipeline
   * Метрики покриття коду тестами