FH University of Applied Sciences TECHNIKUM WIEN

BIF – A

Course: **Software Engineering 1**

Protocol Final Hand-In: MTCG

Executed by:
Edinger Thomas

Reviewer: Fabian Wagner

Protocol for Monster Trading Cards Game (MCTG)

1. Programm Design

The Monster Trading Cards Game (MCTG) server was designed with a layered architecture to separate concerns and improve maintainability:

HTTP Server Layer:

- Manages incoming HTTP requests and routes them to appropriate controllers
- Implements custom handling of HTTP methods and paths without external frameworks
- Utilizes asynchronous programming with async and await to handle multiple client connections efficiently and prevent blocking operations.

Controller Layer:

- Handles business logic for features like user authentication, card management, and battles
- Interfaces with services to fetch or modify data as needed.

Service Layer:

- Encapsulates core functionality such as user management, card acquisition, and battle mechanics
- Acts as an intermediary between controllers and the database layer

Database Layer:

- Uses PostgreSQL for data persistence.
- Managed via a DatabaseManager class to abstract database operations, ensuring modular and testable code

Key Components:

- Deck Configuration: Allows users to set up their battle decks with validation
- Battle System: Implements rules for card interactions, including weather effects and special conditions
- Authentication: Secures API endpoints using tokens generated during login

This design ensures clear separation of responsibilities, facilitating future scalability and maintenance.

2. Technical Decisions

- **Structs for Deck and User:** Deck and User were implemented as structs to allow null values, as Points and ELOs are stored in the database.

- **Database Structure for Cards and Decks**:
  - A user's cards are stored in the cards table, where the user_id column acts as a foreign key referencing the users table. Each card has unique attributes like id and name.
  - A user's deck is stored as an array in the deck column of the users table. This allows for simple storage and management of the player's current card composition.

- **Trade System:** Due to time constraints, the trading system was not implemented in this project.

- **Repository Pattern:** Due to the scope of the project, the repository pattern was not implemented. While beneficial in larger projects, it was deemed unnecessary here to maintain simplicity.

- **Delayed Initialization in YAML:** A delay was introduced in the YAML configuration to avoid errors during the server startup process.

- **Enum-Based Card Display:** Card attributes are returned as enums to simplify processing and ensure UI independence.

- **Scoreboard Filtering:** Administrators are excluded from the scoreboard to prevent skewing the results.

- **Public/Virtual Classes for Testing:** Some classes were made public and/or virtual to enable unit testing, allowing the test project to access internal logic. Therefore, next Time Interfaces will be used.

- **Interfaces:** Initially, interfaces were not used, as they seemed unnecessary. However, this decision complicated unit testing, a learning point for future projects.

3. Unique Feature

A weather system was added to enhance gameplay dynamics by influencing damage calculations during battles. The weather types and their effects are:

- **Sunny:** Boosts Normal-type card damage by 2x
- **Hot:** Boosts Fire-type card damage by 2x
- **Rain:** Boosts Water-type card damage by 2x

The weather changes randomly every 10 rounds during battles. This feature adds a layer of strategy, as players must consider weather effects during battles.

4. Lessons Learned

- **Importance of Interfaces:** Using interfaces would have streamlined the unit testing process. Their absence made mocking and dependency injection very challenging.

- **Docker Experience:** Gained initial experience with Docker, which proved invaluable for managing database containers and ensuring consistent development environments.

- **Handling Larger Projects:** This project highlighted the importance of managing complexity in larger projects, especially in structuring code for maintainability and scalability.

- **Testing Complex Systems:** Learned the value of comprehensive testing to ensure robust functionality across various components and scenarios.

5. Unit Testing

- **Coverage:** 20 unit tests were implemented, covering critical components like battle mechanics, ELO updates, and deck configuration.
- **Critical Tests:**
    - **Damage Calculation:** Verified weather and element effects on card damage.
    - **Battle Outcomes:** Ensured battles resolve correctly with win, loss, and draw scenarios.
    - **ELO Updates:** Confirmed accurate ELO point adjustments post-battle. Since draws occur frequently, it cannot be verified via the provided CURL script whether ELO points are being set correctly in such cases.

- **Challenges:** Mocking complex dependencies required making classes public/virtual, highlighting the importance of initial architectural planning.

6. Time Tracking

Task	Time
Lerning cooding with C#	10 hours
Development	50 hours
Testing	15 hours
Documentation	5 hours
Total	80 hours

7. Git Repository

The complete project source code is available at:

https://github.com/TE-if23b041/MTCG

8. Summary

With significant effort, the project was completed despite its extensive scope. Many components were implemented for the first time, which added to the challenge and learning opportunities. The application runs in a Docker container, which includes a PostgreSQL database to handle data persistence.

This project successfully implemented the MCTG REST API server with key features like user management, battles, a unique weather system, and a scoreboard. Although the trading system was not implemented due to time constraints, the overall functionality provides a robust and engaging user experience. While the omission of interfaces was a notable drawback, the project structure and innovative features highlight the potential for further development and scalability.