Protocol Monster Trading Cards

Julia Pabst

Technical Steps

1. **Initial Setup:**
   - The project was initialized with a base structure using Java, implementing a layered architecture (Controller, Service, Repository).
   - Database connection established via a **ConnectionPool** for efficient query execution.
   - Maven was used as the build automation tool.
2. **Design and Implementation:**
   - Key services (**BattleService, CardService, DeckService,** etc.) were implemented based on the specified requirements.
   - The repository layer handles all database interactions, ensuring separation of concerns.
   - Custom exceptions were created to handle application-specific errors (e.g., **CardsNotFound, TradingDealNotFound**).
   - Controllers were added to handle HTTP requests and responses using a custom server implementation.
3. **Failures and Solutions:**
   - *SQL Injection Prevention:* Prepared statements were used throughout the repository layer.
   - *Trade Logic Issue:* I originally had a different design for database. I had a table for packages and trade table looked differently. So I had to go through all my services and repositories to adapt to the new structure provided below.
   - Duplicate Class Issue: Refactored and removed duplicate implementations in **TradeService**.
   - *Scoreboard Sorting:* Fixed incorrect placement logic for players with the same ELO.
4. **Enhancements:**
   - Added multithreading in **BattleService** to allow concurrent battles.
   - Updated ELO calculation logic to reflect wins and losses dynamically.
   - Designed custom DTOs (Data Transfer Objects) for structured data handling.

Unit Testing Decisions

- **Critical Code Coverage:**
    - **BattleService:** Tested for special rules and battle logic to ensure fair gameplay.
    - **TradeService:** Validated trade constraints and exception handling for trading deals.
    - **DeckService:** Checked for proper deck configuration and validation of card ownership.
    - **ScoreboardService:** Ensured sorting by ELO and correct placement logic.
- **Test Design:**
    - Tests followed the Arrange-Act-Assert pattern for clarity and maintainability.
    - Mocking frameworks (Mockito) were used for isolating dependencies.
    - Example battle unit tests
        - The unit tests for **BattleService** were designed to ensure fairness in battles by validating the application of special rules, such as "Goblin is afraid of Dragon" and "Wizzard controls Ork."
        - The test cases include scenarios with regular battle mechanics and edge cases, such as ensuring no endless battles by limiting the rounds to 100.
        - Mocked services (e.g., UserService and CardService) isolate dependencies, allowing tests to focus solely on battle logic and game outcomes.
        - Coverage includes verifying ELO updates for winners and losers, ensuring proper score adjustments and fair game progression.
        - The chosen tests aim to simulate real-world gameplay scenarios, ensuring reliability and accuracy in critical game functions while improving player experience.

Lessons Learned

- Proper exception handling and meaningful error messages significantly enhance debugging: Before doing that I was always lost and it took me ages to find mistakes.

In addition to that, it really helped using the debugger to go through the code step by step.

- Early adoption of testing practices prevents cascading issues in dependent modules: By starting to use unit tests from an early stage, I had immediate feedback on if my implementations were valid or not.
- Effective use of version control (e.g., Git) simplifies change tracking: Without the possibility to go back to a previous working version of my code, I would have been lost. In future projects I plan to commit even more often to be able to find bugs more efficiently.
- Mistakes in early planning can have severe downstream consequences: A flawed initial database design caused significant additional work as I had to detangle everything I have worked on before. In the future, I will seek feedback from my supervisor or teacher earlier to mitigate such risks.
- Systematic testing improves overall code quality and ensures critical functionalities work as intended before integration: Adopting tests at an early stage helps creating a streamlined development process.

Unique Feature

- Elo calculation is calculated dynamically and if several players have the same elo, they are at the same position in the scoreboard.

Project Design

- **Structure:**
  - Layered architecture with clear separation of concerns.
  - Entities represent core game components like **Card, User, Trades.**
  - DTOs ensure structured communication between layers.
- **Class Diagram Highlights:**
  - **BattleService:** Handles battle logic and ELO updates.
  - **TradeService:** Manages trade creation and validation.
  - **CardService:** Centralized card-related operations, including ownership and deck management.

Tracked Time

Unfortunately, I forgot to track my time early on. The declarations below are based on approximate assumptions in hindsight.

- Initial Setup (HTTP Server) and Database Design: 13 hours
- Core Feature Implementation: 20 hours
- Debugging and Refactoring: 10 hours
- Unit Testing: 8 hours
- Documentation and Protocol Writing: 1 hours

What I can guarantee is the number of commits I did in each month

- November: 9 commits
- December: 37 commits
- January: 10 commits

Link to Git Repository The full project and commit history can be found at:

https://github.com/JuliaPabst/Monster-Trading-Cards-Game