Assignment 1

Domain Layer Implementation

In this assignment you will implement the **domain** layer (as modeled in the previous assignment), and the **service** layer. The service layer is used to initiate the different use-cases.

Additionally, you are presented with new functional and non-functional requirements. Adjust and update your models and documentation as necessary to accommodate these changes.

Implementation Guidelines

Team Roles

At each version the team decides who takes the role of project manager, and the role of acceptance tester. The rest of the team members are developers. All team member must experience both roles during the course. This means that each version has a 4-person development team (5 if you're 6 in a group) + an acceptance tester.

Responsibilities of the project manager

- Lead the team meetings
- Define tasks.
- Prioritize them.
- Estimate their time to completion.
- Split them wisely among the team members.
- Track the progress and make adaptations when needed.
- Track the use cases through the business requirements, to the technical requirements, to the code level, see that there are tests that cover the use case paths.
- Develop the models, and maintain them in sync with the implementation.
- Keep the acceptance tester from "mingling" with the developers.

Responsibilities of the acceptance tester

- First and foremost: **not** work on implementation. Ideally, the acceptance tester should have absolutely no idea what the developers are doing, to keep the acceptance tests strictly customer-facing.
- Construct the acceptance testing infrastructure as described below.
- Write acceptance tests for different scenarios for every use-case.
- Write acceptance tests for non-functional requirements.

Responsibilities of each developer

- Write and run unit tests for **their own** code.
- Follow coding and testing conventions.
- Boy Scout rule: leave the code better than how they found it.

Required Tools

You must use version control and project management tools. However, you may choose the ones you will work with.

Source Control Workflow

Note: The language used in this section is very git-centric, but the ideas apply to any source control tool you choose.

In this assignment you will be writing lots of code, and most probably more than one person will write code at any given time. The project manager has the additional role of “editor-in-chief”, and is the only one allowed to commit to the master branch.

All team members work on feature branches, also known as topic branches. These are short-lived branches which are used to implement a single feature, fix a single bug, and other focused tasks. When work on a feature branch is done, the developer will issue a pull-request and notify the project manager.

The project manager will approve the request if:

1. Without talking to the developer - she/he can understand the purpose of new/modified methods and classes (based on naming conventions and documentation).
2. She/He believes that the code is written well.
3. She/He believes that all needed tests have been written to check the code correctness.

If the project manager declines the request, the decline will be accompanied by follow-up tasks to fix it. If the request is approved, the project manager will merge it to the master branch. At that point, other developers should pull the latest master branch so their codebase is up-to-date.

This workflow is customary in many software companies as it helps keeping the code clean, readable, and most important correct. It will endow you with great programming habits.

This workflow is **mandatory**.

In order to understand better, we encourage you to watch this short tutorial - The Art of Code Review.

Functional Requirements

In addition to implementing all use-cases from the modeling assignment, you will also implement the following use-cases:

- Edit user profile
    - Change password / email
- Create game
    - Set game preferences
        - Game type policy: limit, no limit, pot limit
        - Buy-in policy – the cost of joining the game.
        - Chip policy – determine the amount of chips each player is given (a value of zero means the game is played with real currency).
        - Choose minimum bet (equals to the big blind).
        - Define minimal and maximal amount of players in the table.
        - Choose whether spectating a game (viewing the game without playing) is allowed or not.
- Search / filter active games
    - By player name
    - By pot size
    - By game preference
- Leagues - the highest ranking user in the system may:
    - Set a default league for new users
    - Set criteria for moving to a new league
    - Move users between leagues

Non-Functional Requirements

Unit / Integration Testing Infrastructure

Starting from this assignment, you will manage an automated testing infrastructure for your unit, and integration tests. This infrastructure should allow you to run some, or all, of the tests after modifying your code, and easily add and remove tests. You and your tests should aspire to achieve 100% code coverage.

For each test type, decide how its tests should be organized, e.g.:

- Separate test suites vs. one big suite
- Keeping the tests alongside the code vs. keeping them separate

Acceptance Testing Infrastructure

Unlike unit and integration tests, acceptance tests should act as “executable specifications”: they should be readable and understandable to the customer, mirror the different requirements, and most importantly, be completely oblivious to the system’s implementation.

Naturally, these tests will be separate from implementation code, and are completely “black-box”.

Both Infrastructures

Each test should contain information on what it tests, and why is it required. There are numerous naming conventions for test methods and test classes. Pick one, and stick with it. An example can be found here.

Keep your tests under version control, since they will change alongside the requirements.

Error Handling

Logging

You must maintain a log which will track **all** errors in the system. Note that “error” does not necessarily mean an exception that is “thrown” or “raised” by your runtime environment, but any situation which counts as invalid in our domain.

Some guidelines:

- Tag entries with their severity / priority
- Provide enough information to understand what went wrong, and where
- Avoid storing entire stack traces directly in the log (more verbose than useful)

Exception Handling

As detailed in the second lecture slides.

Required Models and Specifications

- Add the service layers to the component diagram.
    - Now is a good time to decide whether the domain layer is fully exposed to the service layer or exposes a set of interfaces.
- Draw sequence diagrams for the following use-cases:
    - Save favorite turns from replays in order to be viewed on later occasions
    - Set criteria for moving to a new league
    - Spectate active game

    - Store all the information from a game, such as: actions performed by all players in the game, the cards dealt at each round, round beginning and end, etc.
    - Support playing a Texas Hold'em game: dealing cards, placing blind bets for players, allowing players to check (NOP), fold or bet according to the game rules, etc.
    - Find all active games which the user can join.

  These use-cases and their sequence diagrams will follow us until the final version, and in each version you will update them and show how they evolved with the addition of new components to the system.

- All diagrams - updated with any changes from this version:
    - Component diagram.
    - Class diagram showing the primary objects in the domain logic, divided by their components. Each element must include its primary methods (i.e., methods that explain what the object does. No getters, setters, helpers, etc.). The diagram should not contain “auxiliary” classes, e.g. factories, builders, etc.
- Obviously, your implementation must match your model and vice versa - so keep them synced!

Deliverables

The project manager will present the following:

- Version summary, including:
    - The status of each use-case: is it implemented or missing? Was it changed in relation to the previous assignment? What was the change?
    - Who did what
- All diagrams and specifications as defined in the previous section
- The codebase structure
- Tests run: running unit, integration, and acceptance tests for all use-cases, and showing coverage statistics.

Each team member must understand all models thoroughly (i.e. diagrams, not use-cases). In addition the team members will present their part in the assignment.

Class Presentation

A team member will prepare a 10-15 minute discussion on the topic of mock and stub objects. The discussion should cover:

- What are mock and stub objects.
- The differences between them and when to use each.
- Compare test maintainability with automatic and manual mocks.

- Disadvantages and common pitfalls.

Choose a mocking framework compatible with the language of your choice which you will use from this assignment to the end of the project, and demonstrate its usage in your project's test suite. The demo must include creating a mock, setting expectations, and verifying the expectations set on the mock.

Please refer to the presentation guidelines at the Assignment page.