Ping-Pong Tournament MVC

Analysis and Design Document

Mureșian Dan-Viorel
30433

Table of Contents

1. Requirements Analysis

1.1. Assignment Specification

Since the Ping-Pong Tournaments application is such a big success, the owners of the application wish to add a new feature to it: Paid Tournaments. They differ from the free tournaments available until now by the fact that they require an enrollment fee and offer a cash prize to the winner. For the moment, the player that finishes on the 1st position receives all the money from the enrollment fees of the tournament. From the beginning of the tournament until a player wins 1st place, the prize money is kept in the account of the Ping-Pong Association or in the account of the Tournament itself. Based on Assignment A1, adapt your application to fit the new requirements.

1.2. Functional Requirements

The regular user can perform the following additional operations:

- Enroll into upcoming Tournaments, by paying the enrolment fee out of their account;
- View Tournaments by category: Finished, Ongoing, Upcoming;
- Search Tournaments by name and type (free/paid).

The administrator can perform the following additional operations:

- CRUD paid tournaments;
- Add money to any player's account;
- Withdraw money form any player's account.

Application Constraints and Technical Requirements:

- Use the Model View Controller Pattern for all views;
- The Data Access Layer (DAL) will be re-implemented using an ORM framework;
- The application will use a config file from which the system administrator can set which DAL implementation will be used to read and write data to the database;
- Use the Abstract Factory design pattern to switch between the new and the old DAL implementation (implemented with JDBC or equivalent);
- All the inputs of the application will be validated against invalid data before submitting the data and saving it. (e.g. Tournament start date, Mandatory fields, The player has enough money in his account, etc.);
- Write at least one Unit Test for each new method in the business layer (e.g. Test that you can transfer money from the Player's account to the account of the Ping-Pong association).

2. Use-Case Model

Use Case: Modify Own Score
Level: player level
Primary Actor: player
Main success scenario:

- Start application;
- Click the Login button;
- Login with email and password;
- Click the View Tournaments button;
- Select the tournament you are taking part in;
- Click the View Tournament button;
- Click the Refresh button to populate the list;
- Double click on your name to get the match list;
- Select the current ongoing game;
- Click View Game;
- Use + for increasing the score, - for decreasing it;
- Click Update Score when you are done;
- Click Refresh to see updates;

Extensions:

- Once a game has finished, the score can't be modified anymore;
- Once a match has finished, the score in a game can't be modified anymore;
- Clicking on another player's match will generate an error.
- Invalid Login credentials;

Use Case: Create Tournament
Level: administrator level
Primary Actor: administrator
Main success scenario:

- Start application;
- Click the Login button;
- Login with an administrator account;
- Click the View Tournaments button;
- Click the Create Tournament button;
- Enter tournament data and click Create Tournament;

Extensions:

- Entering null data will generate an error;
- You cannot create an account without an administrator account;
- Invalid Login credentials;

Use Case: View matches
Level: player level
Primary Actor: player
Main success scenario:

- Start application;
- Click the Login button;
- Login with email and password;
- Click the View Tournaments button;
- Select the tournament you are taking part in;
- Click the View Tournament button;
- Click the Refresh button to populate the list;
- Double click on your name to get the match list;

Extensions:

- Invalid Login credentials;

Use Case: View tournaments
Level: player level
Primary Actor: player
Main success scenario:

- Start application;
- Click the Login button;
- Login with email and password;
- Click the View Tournaments button;

Extensions:

- Invalid Login credentials;

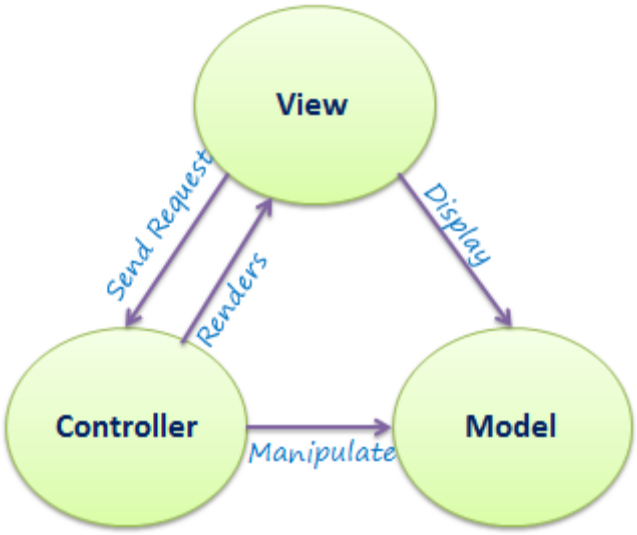
3. System Architectural Design

3.1. Architectural Pattern Description

Model View Controller or MVC as it is popularly called, is a software design pattern for developing web applications. A Model View Controller pattern is made up of the following three parts:

- **Model** − The lowest level of the pattern which is responsible for maintaining data;
- **View** − This is responsible for displaying all or a portion of the data to the user;
- **Controller** − Software Code that controls the interactions between the Model and View.

MVC is popular as it isolates the application logic from the user interface layer and supports separation of concerns. Here the Controller receives all requests for the application and then works with the Model to prepare any data needed by the View. The View then uses the data prepared by the Controller to generate a final presentable response. The MVC abstraction can be graphically represented as follows.



The Model

The model is responsible for managing the data of the application. It responds to the request from the view and it also responds to instructions from the controller to update itself.

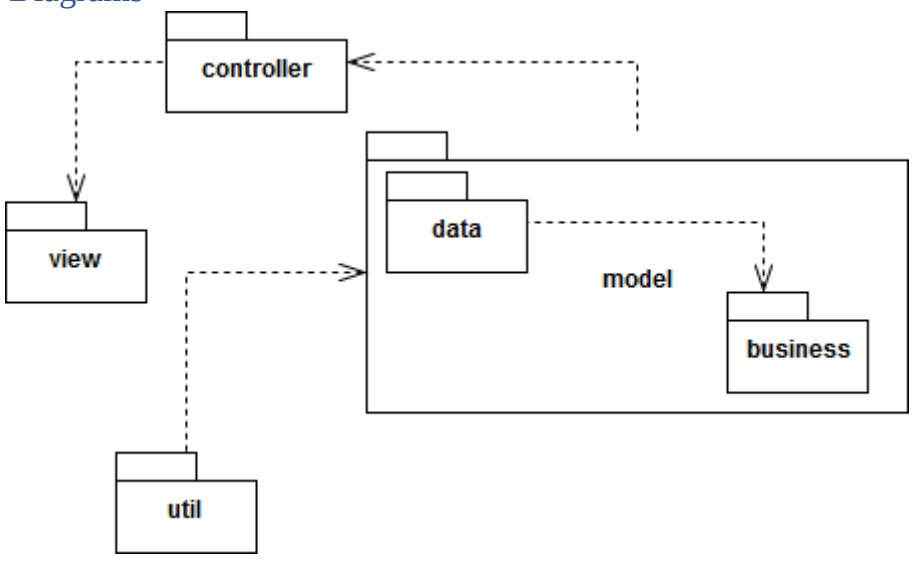
The View

It means presentation of data in a particular format, triggered by a controller's decision to present the data. They are script-based templating systems like JSP, ASP, PHP and very easy to integrate with AJAX technology.

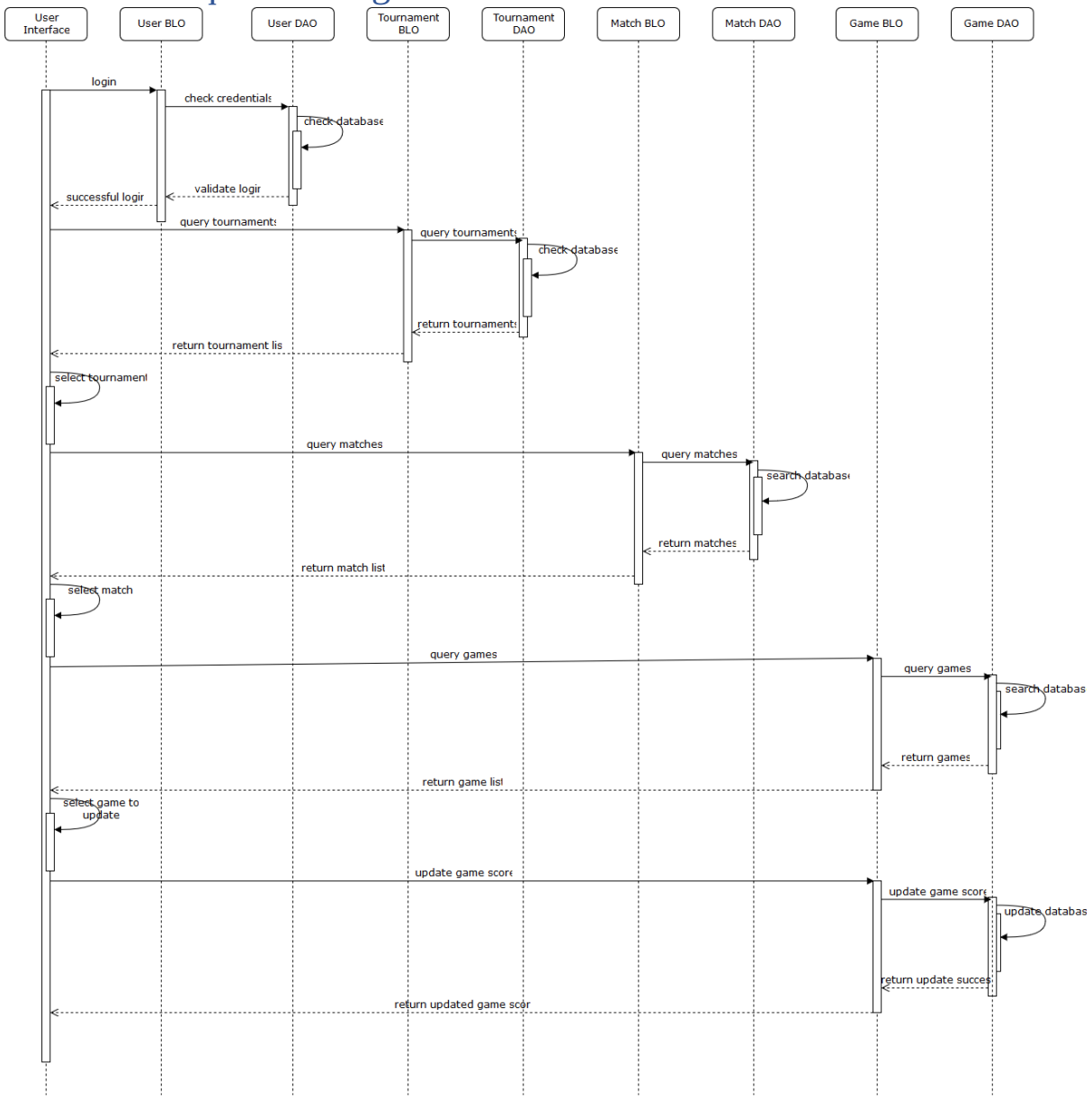
The Controller

The controller is responsible for responding to the user input and perform interactions on the data model objects. The controller receives the input, it validates the input and then performs the business operation that modifies the state of the data model.

3.2. Diagrams



4. UML Sequence Diagrams

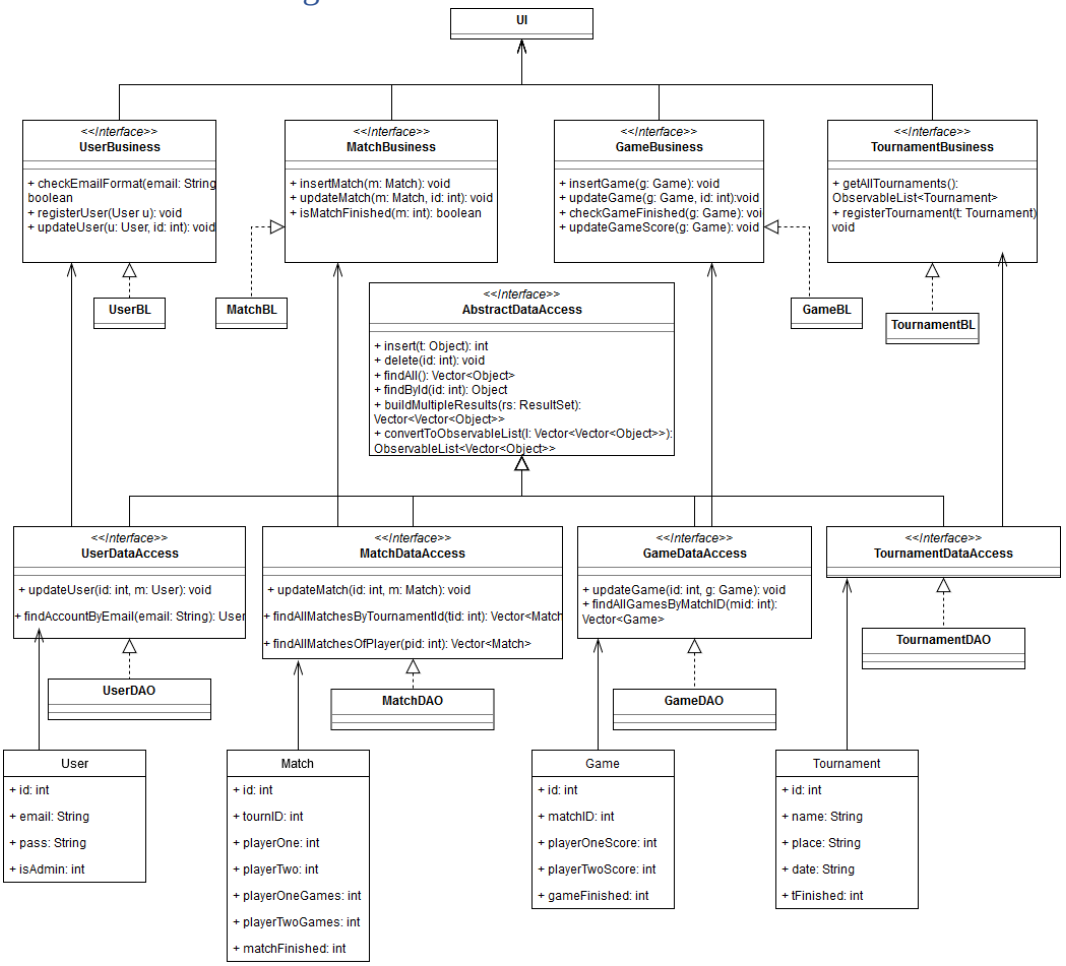


5. Class Design

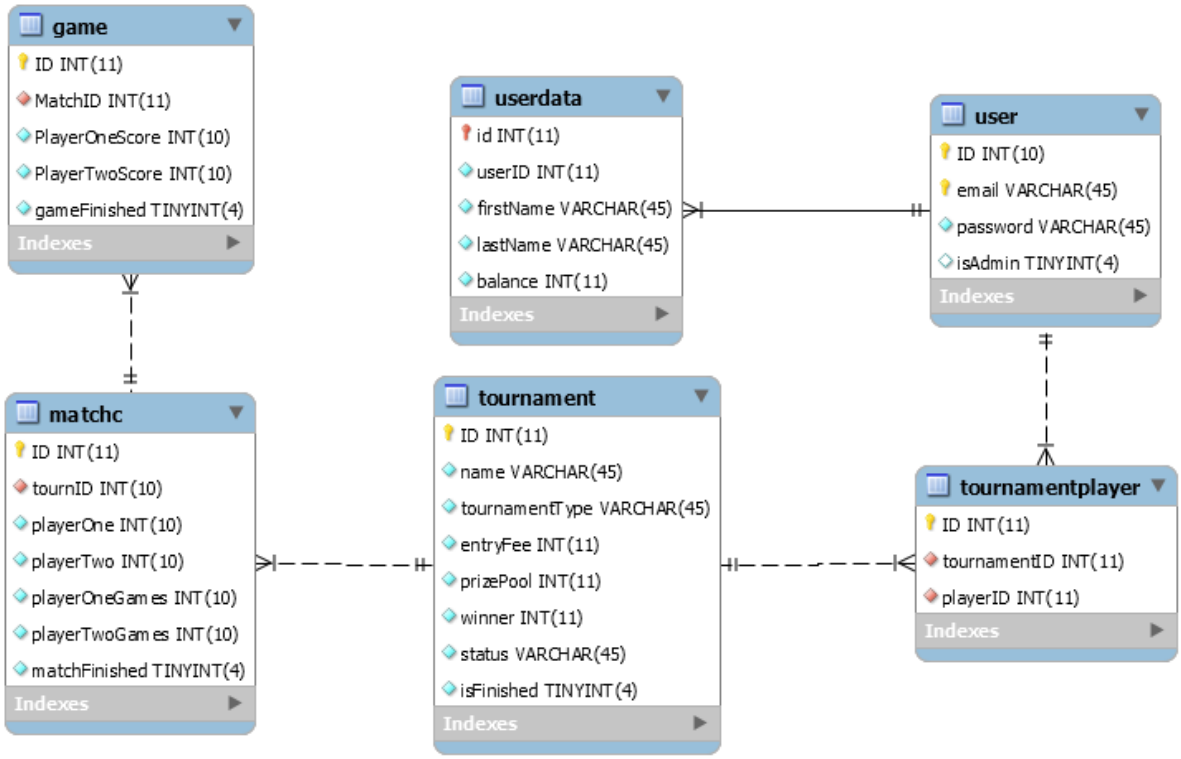
5.1. Design Pattern Description

The Abstract Factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes. In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. The client doesn't know (or care) which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.

5.2. UML Class Diagram



6. Data Model



7. System Testing

7.1. White-box Testing

White-box testing uses the control structure of the procedural design to derive test cases that (i) guarantee that all independent paths within a module have been exercised at least once, and that (ii) exercise all logical decisions, all loops and internal data structures. Control structure testing is an example of white-box testing method which includes the following:

a) Condition testing

- Exercises the logical conditions in a program module;
- If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include the following:
  - Boolean operator error (incorrect/missing/extra Boolean operators);
  - Boolean variable error;
  - Boolean parenthesis error;
  - Relational operator error;
  - Arithmetic expression error.

- Strategies:
    - *Branch testing* - for a compound condition C, the true and false branches of C and every simple condition in C need to be executed at least once.
    - *Domain testing* - for E1 <relational-operator> E2, 3 tests are required to make the value of E1 greater than, equal to, or less than that of E2.
    - *Branch and relational testing* - detects branch and relational operator errors in a condition if all Boolean variables and relational operators in the condition occur only once and have no common variables.

a) Loop testing

7.2. Black-box Testing

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software and enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques; rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods. Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external data base access, (4) behavior or performance errors, and (5) initialization and termination errors. Black-box testing includes methods such as the following:

a) Equivalence partitioning

- Divides the input domain of a program into classes of data from which test cases can be derived.
- An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:
    - If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
    - If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
    - If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
    - If an input condition is Boolean, one valid and one invalid class are defined.

b) Boundary value analysis

- Complements equivalence partitioning by exercising bounding values. Rather than selecting any element of an equivalence class, the boundary value analysis leads to the selection of test cases at the "edges" of the class.

8. Bibliography

1. https://pongworld.com/table-tennis-sport/rules;
2. https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller;
3. https://www.journaldev.com/2954/hibernate-query-language-hql-example-tutorial;
4. https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html;
5. https://en.wikipedia.org/wiki/Abstract_factory_pattern;
6. https://www.journaldev.com/3793/hibernate-tutorial.