Server architecture of an online multiplayer card game

Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brasil

Abstract

Pirates is a turn-based card game for 2 to 6 players that consists of competing to win treasures, where the first player that collects 5 treasures first wins the game. The current work intends to present an architecture of an online multiplayer server and evaluate from a software quality point of view the design choices, code structure, versioning and naming standardization practices. An action system was created to allow reuse and creation of new expansions for the game. The objective of the work is to evaluate the flexibility, testability, readability and scalability.

CCS Concepts: • ;

Keywords: quality, distributed systems, tests, refactoring

ACM Reference Format:
. 2022. Server architecture of an online multiplayer card game. In *Proceedings of UFMG Workshop on Software Engineering (Conference WSE).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Pirates is a turn-based card game for 2 to 6 players that consists of competing to win treasures, where the first player that collects 5 treasures first wins the game. The current work intends to present an architecture of an online multiplayer server and evaluate from a software quality point of view the design choices, code structure, versioning and naming standardization practices. An action system was created to allow reuse and creation of new expansions for the game. The server consists of a layered architecture aiming to isolate the game's business rules in order to facilitate unit testing of each entity in the domain. An action system was implemented, it allows defining a flow of operations and player choices in order to reuse and isolate code in its appropriate responsibilities. A protocol was also developed to isolate completely the clients from server game rules, converting complex rules flows into simple choices for the client, This is important as it increases the security of the game, reducing the possibility of cheaters breaking any rules of the game. game, keeping the rule only on the server also makes it easier to process unit tests from the beginning to the end of the game, increasing the safety of implementing new functionalities. The objective of the work is to introduce architecture developed, presenting its advantages in order to facilitate the implementation of new game mechanics.

Conference WSE, Dezembro, 2022, Belo Horizonte, MG

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

2 ARCHITECTURE

The server follows a layered architecture with a focus on isolating the domain from the rest of the layers. This model is suitable as it facilitates scalability and layer changes by changing only the previous layer. Despite the online multiplayer project, it is possible to replace layers in case the game goes offline against artificial intelligence without changing the existing domain.

It is important to point out that the protocol is presented within the server structure as a namespace to facilitate testing the server with the client on the command line. The end result of this namespace is to become a private package completely isolated from any project, so both the client and the client must add this package as an external dependency to obtain the classes necessary for communication.

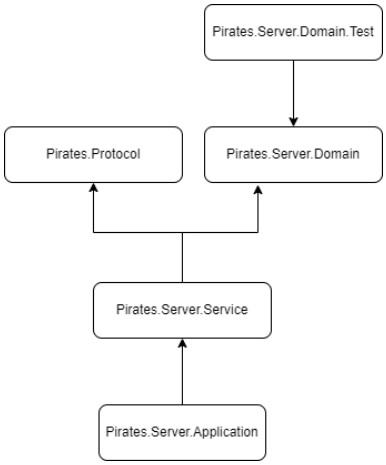

Figure 1. Namespaces diagram

The architecture was designed to have all the rules processing logic on the server, isolating the client from any knowledge of the game's domain. The main purpose of this decision is to serve as a barrier that tries to prevent cheating in the game. The authors [3] and [4] classify this approach as server-focused and present further advantages for this model,

such as reducing client-side processing and ease of modifying application rules. domain without necessarily having to generate a new build and make a new customer publication.

The naming structure, code formatting and semantics were followed based on the concepts presented by [2] with the aim of making the code easier to read for new developers and facilitating the maintenance of the code structure in the long term. To ensure standardization of the structure among multiple developers, 'Editor Config' is used, a code formatting technology that allows the creation of configuration files that describe the correct structure to follow. Currently, most integrated development environments and text editors supports it.

2.1 Domain

The system isolates all business rules through a dedicated 'namespace', this space is dedicated to processing game rules. This isolation allows for greater organization of the domain and facilitates the creation of tests of unity. Another advantage is the possibility of changing the nature of the system to an offline game without having to change any code in this space.

2.1.1 Main classes. The main domain classes are 'Player', 'Table', 'BaseCard' and 'BaseAction' in order to represent the basic structure of the game where the player performs an action involving a card on the table.

The 'Table' class is where all actions performed by players are executed, it is responsible for dictating the entire gameplay flow, having all the necessary elements for the game to run similar to what was implemented by [**?**] classifying the 'Gameplay Engine' pattern.

2.1.2 Action system. Player interaction consists of an action system that groups choices and operations to be performed at the table, and each type of action has its purpose well defined in the system. Actions can be chained together with the aim of performing a certain flow. The figure **??** shows a UML diagram with the main entities.

- Primary: Main actions that the player can perform on the turn.
- Resultant: Actions resulting from the execution of another action and that require some choice from the player.
- Passive: Action to be performed at the end of the player's turn generated by some card on the field.
- Immediate: Action to be performed immediately after being returned by another action.

The duel flow is the most complex in the game, and as such, it is the best flow to exemplify how the action system solves the communication and decision problems presented by it. The flow shown in the figure **??** of the game consists of a player choosing a duel initiator card and deciding which player to start the duel with. After the duel begins, the target player must play a duel card face down. Both players can play surprise duel cards to increase the chance of winning the duel. The players must then turn over their duel cards and whoever has the most duel points is declared the winner and can steal a card from the loser's hand.

The server implements the previous flow using the system of actions illustrated in the figure /refdiagrama-atividade-sistema-acao-duelo as follows: When executing the primary action "Duel", the server returns the resulting action "ChoosePlayer", after choosing who to duel with, the server returns the "ChooseStartingCard" action so that the player chooses which card in his hand to start the duel, after making this choice the server sends the "DrawDuelAnswerCard" action for the target player to play his response duel card. The server checks if any of the players have a surprise duel card and sends the "DuelSurpriseCardDown" action to those who do. The server then executes the immediate action "CalculateDuelResult" to calculate the winner and loser of the duel and sends the resulting action "StealCard" to the winner.

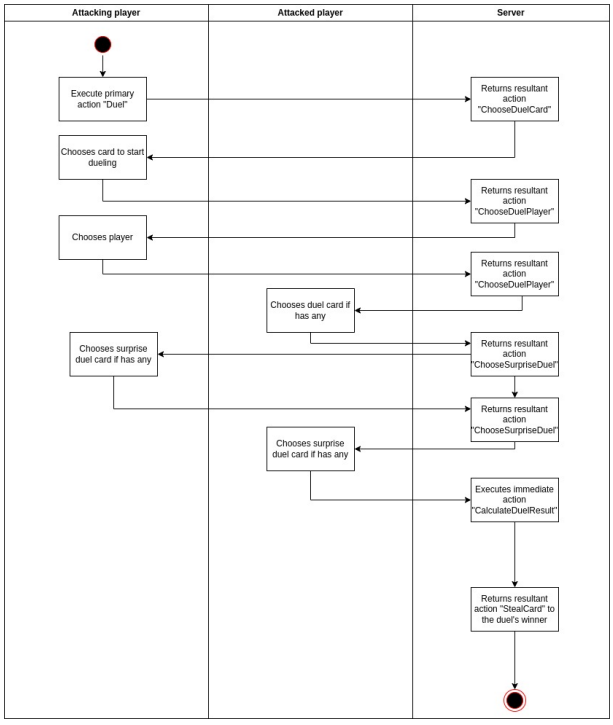

Figure 2. Activity diagram showing how the action system works in the duel flow

2.1.3 Unit tests. Unit tests are implemented on top of all classes in the project domain. The folder structure is analogous to that of the domain to facilitate navigation.

To facilitate writing the tests, the NSubstitute test mock package was used. It creates at runtime a class that inherits the generic parameter passed and allows you to define what is returned in each method and property without the need

for formal invocation of the constructor and other methods of the class.

Listing 1. Rum unit test

```
namespace Pirates.Server.Domain.Test.Card.ImmediateResolution;

using System;
using System.Collections.Generic;
using Action;
using Deck;
using Domain.Card;
using Domain.Card.ImmediateResolution;
using NSubstitute;
using NUnit.Framework;

public class RumTests
{
    private Table _table;

    public RumTests()
    {
        var cardsConfiguration =
            new List<Tuple<string, int>> {new(nameof(Rum), 1)};

        CardsGenerator.Configure(cardsConfiguration);
    }

    [SetUp]
    public void SetUp()
    {
        var players = new List<Player>();

        var player1 = new Player(
            "player1",
            (_, _) => { },
            (_, _) => { },
            (_, _) => { },
            (_, _) => { });

        var player2 = new Player(
            "player2",
            (_, _) => { },
            (_, _) => { },
            (_, _) => { },
            (_, _) => { });

        var player3 = new Player(
            "player3",
            (_, _) => { },
            (_, _) => { },
            (_, _) => { },
            (_, _) => { });

        players.Add(player1);
        players.Add(player2);
        players.Add(player3);

        _table = new Table(players);
    }

    [Test]
    public void ApplyEffectMustBuyCardsToPlayer()
    {
        var cardsAtHand = new List<Card>();

        var starterPlayer = new Player(
            string.Empty,
            null,
            null,
            null,
            null);

        var cardsOnCentralDeck = new List<Card> {Substitute.For<Card>(), Substitute.For<Card>()};

        _table.CentralDeck.PushTop(cardsOnCentralDeck);
        starterPlayer.Hand.Add(cardsAtHand);

        var action = Substitute.For<BaseAction>(starterPlayer, null);

        var rum = new Rum();

        rum.ApplyEffect(action, _table);

        foreach (Card card in cardsOnCentralDeck)
            Assert.IsTrue(starterPlayer.Hand.Exists(card));

        Assert.IsNull(_table.CentralDeck.GetTop());
    }
}
```

The unit testing package used was NUnit due to its wide adoption by the community, making it easier to obtain answers to questions.

2.2 Client server communication

2.2.1 Web Socket. Web socket is a communication protocol built on a continuous TCP connection that allows active sending of packets both by the client and the server. Facilitating the transport of real-time data to multiple customers connected. [1] This protocol is used to implement a browser-based multiplayer game via bidirectionality, in order to allow the server to send messages to the client without the client making a request asking for that message.

The server uses this protocol because it makes it easier to send packets to all players, allowing the server to notify players when a card is removed from a player's hand, for example. It is mentioned by [4] the idea of using TCP as a communication protocol mainly because it is responsible by ordering the packages, a point that is essential for a card game. Considering that the communication flow of the game is considerably smaller than real-time game servers, the cost of this protocol does not present relevant to the current solution.

2.2.2 Protocol. A protocol was developed to standardize communication and isolate domain knowledge among clients. They exist client and server packages for the two implemented context types: Room and Match.

The communication flow consists of sending a server message and the client responding to the received message.

Server Room Package:

- Id: Package ID.
- ErrorId: Id of the error if it occurred.
- ErrorDescription: Description of the error if it occurred.
- HasError: There is an error in the package. Made to facilitate error checking.
- PlayerId: Player ID that receives the package.
- IdRoom: Id of the room in which the operation took place.
- MatchId: ID of the match if it has been initialized.
- StarterPlayerId: Id of the player who performed the operation.
- RoomOperationType: Type of operation performed (enter, exit, start departure).

Listing 2. server room packet JSON

```
{
```

```
    "Id": "91b7388b-6a53-4df5-bad2-562d1e98f610",
    "ErrorId": "",
    "ErrorDescription": "",
    "HasError": false,
    "PlayerId": "2536612e-84be-46c6-879c-d9e8755fbfa3",
    "RoomId": "ba1b1b64-08a6-4a45-b955-eeb9e9f39bd0",
    "MatchId": "",
    "StarterPlayerId": "1adeb4ef-9c90-46df-b61b-9c57b90ebd87",
    "RoomOperationType": 1
}
```

Customer Room Package:

- Id: Package ID.
- PlayerId: Player ID who receives the package>
- IdRoom: Id of the room in which the operation took place.
- StarterPlayerId: Id of the player who performed the operation.
- RoomOperationType: Type of operation performed (enter, exit, start departure).

Listing 3. JSON do pacote da sala do cliente

```
{
    "Id": "ba1b1b64-08a6-4a45-b955-eeb9e9f39bd0",
    "StarterPlayerId": "ffa93170-17d5-4118-9ad5-0afdc12c8fb4",
    "RoomId": "2536612e-84be-46c6-879c-d9e8755fbfa3j",
    "RoomOperationType": 1
}
```

Server Starter Package:

- Id: Package ID.
- ErrorId: Id of the error if it occurred.
- ErrorDescription: Description of the error if it occurred.
- HasError: There is an error in the package. Made to facilitate error checking.
- Events: Events that occurred on the table. (Card added, removed from player's hand, etc.).
- Treasures: Number of treasures the player has.
- RemainingActions: Number of remaining shares of the player.
- TableId: Id of the table.
- DateTime: Date and time of the package creation.
- Choice: Choice for the player to make (choice between cards, actions, etc.).
- StarterPlayerId: Id of the player who performed the action

Listing 4. server packet match JSON

```
{
    "Id": "dadbfc3c-bff6-4970-aa23-c2c78eeac721",
    "HasError": false,
    "ErrorId": "",
    "ErrorDescription": "",
    "RemainingActions": 2,
    "Treasures": 1,
    "Events":
    {
        "888b4316-f6da-4e46-93f9-afc8df092763":
        [
            {
                "Place": 3,
                "CardId": "rum",
                "Added": false
            }
        ]
    },
    "TableId": "a61dab79-3046-4072-a229-9821f5d620b5",
    "StarterPlayerId": "767507ef-3784-440a-b86b-e0a8b707611f",
    "DateTime": 1667923144,
    "Choice":
    {
        "Options":
        [
            "rum",
            "parrot",
            "kraken"
        ],
        "ChoiceLimit": 1
    }
}
```

Customer Departure Package:

- TableId: Id of the table.
- StarterPlayerId: Id of the player who performed the action.
- DateTime: Date and time of the package creation.
- ExecutedActionId: Id of the action the player is executing.
- Choice: Choice answer.

Listing 5. client match packet JSON

```
{
    "TableId": "9b315933-487e-4ef0-8cc1-85c9f2ee9cee",
    "StarterPlayerId": "ffa93170-17d5-4118-9ad5-0afdc12c8fb4",
    "DateTime": 1667923144,
    "ExecutedActionId": "choose-card",
    "Choice": "parrot"
}
```

2.3 Service

This layer is responsible for serving as an intermediary between the domain and the application layer, isolating the knowledge of one from the other. This layer has numerous services with different purposes but which serve as the layer that deals with the project's "technology", i.e., reading/writing files, accessing databases, web requests, among others.

2.3.1 Configuration. Responsible for obtaining the configuration file in JSON format and converting it into a domain instance so that services and domains can configure themselves accordingly. The project follows the pattern of creating a configuration file per environment using the following naming pattern: configuration.<ENVIRONMENT>.json

2.3.2 Log. The logging service is responsible for configuring the Serilog log package to simultaneously write to a text file for future analysis and to the console. The service also ensures that unhandled exceptions are caught and properly logged from the "AppDomain.CurrentDomain".

2.3.3 Room. Responsible for managing all open rooms and processing transactions made by customers. Notifies the match manager when a new one should be started.

2.3.4 Match. It consists of a service responsible for receiving messages from the client, forwarding them to the table instance and making the necessary conversions of the domain result to the protocol structure.

2.3.5 Match manager. Responsible for storing ongoing matches. This service acts as an intermediary between the match controllers and the match service itself. It is also responsible for creating matches when a room is complete and removing it when the game ends.

3 FUTURE WORKS

3.1 Tests with real matches

It is possible to persist all client/server communication from valid matches and reprocess this data, comparing the match input and output data persisted with what the server just processed. Considering that the server must work deterministically, this solution allows you to run infinite real matches on the server, testing whether the server remains consistent when it comes to match processing. This solution can increase the code security in order to increase the guarantee that the flow that already exists has not been broken by implementations new ones. It is worth mentioning that there is a scenario in which introduced functionalities change existing flows, thus Therefore, the tests should not run non-compatible games.

3.2 Optimization of memory usage of card instances

Currently the server creates a card instance for each card available in the game. However, this instance does not have nothing different from an instance of the same card from a player at another table. This way, the server wastes memory. A possible solution would be to create an instance of each card and only handle card IDs in the deck and hand of the players and, when you need to execute the effect of a card, ask that entity to execute it based on your id. This allows only one instance of each card to exist for all active tables on the server.

4 CONCLUSION

Pirates is a turn-based card game for 2 to 6 players that consists of competing to win treasures, where whoever Getting 5 treasures first wins the game. The present work intends to present an architecture of a server online multiplayer and evaluate from a software quality point of view the design choices, code structure, versioning and naming standardization practices. The server consists of a layered architecture with the aim of isolation of the game's business rules in order to facilitate unit testing of each entity in the domain. Main The objective of the work is to evaluate the architecture and modality developed for the game, in order to verify the flexibility implementation of existing flows and possible new flows to be introduced by expansions with new cards and mechanics. The work presents the action system as the main mechanism for easing the flow structure to control game events and is rated satisfactory for supporting new mechanics. Another positive point is the protocol because it allowed the complete isolation of clients from the game's business rule based on the abstraction of choices. Despite having multiple points of design improvements and memory and processing optimization, the server is suitable for entering a production environment.

References

[1] Bijin Chen and Zhiqi Xu. 2011. A framework for browser-based Multiplayer Online Games using WebGL and WebSocket. In *2011 International Conference on Multimedia Technology*. 471–474. https://doi.org/10.1109/ICMT.2011.6001673

[2] Robert C. Martin and James O. Coplien. 2009. *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, Upper Saddle River, NJ [etc.].

[3] D. Varun Ranganathan, R. Vishal, Vallidevi Krishnamurthy, Prashant Mahesh, and Roopeshwar Devarajan. 2017. Design patterns for multiplayer card games. In *2017 International Conference on Computer, Communication and Signal Processing (ICCCSP)*. 1–3. https://doi.org/10.1109/ICCCSP.2017.7944107

[4] Cai Lan Zhou, Yu Kui Wang, and Sha Sha Li. 2011. Design of server for network casual games. In *2011 International Conference on Computer Science and Service System (CSSS)*. 2193–2196. https://doi.org/10.1109/CSSS.2011.5972196