

Dokumentacja Projektu “Brydż”

Hubert Krata

PO 2025

1 Ogólna architektura

Projekt zrealizowaliśmy w architekturze MVC.

Opis komponentów, którym odpowiadają pakiety:

- **model** — zawiera klasy modelujące logikę gry (w szczególności **Game**, **Bidding** i **Scoring**)
- **controller** — zawiera logikę działania klienta (event handlers GUI, odbieranie wiadomości, itp). Jeśli klient hostuje serwer, trzyma referencję do obiektu **Server**, ale nic z nią nie robi.
- **view** — zawiera klasy odpowiadające za wygląd GUI. Zastosowaliśmy framework JavaFX.
- **server** — jedyną publiczną klasą jest **Server**, odpowiadająca za... serwer.
- **communication** — klasy ułatwiające komunikację klient-serwer.

2 Nieco bardziej szczegółowe opisy niektórych komponentów

2.1 Model

Podstawowe klasy modelu:

- **Bidding** — odpowiada za licytację.
- **Game** — odpowiada za przebieg rozdania (w szczególności, rzucanie kolejnych lew). Zawiera referencję do **Bidding**.
- **Scoring** — odpowiada za liczenie wyniku z wielu gier.

Istnieją testy jednostkowe do modelu.

2.2 Communication

Dzięki pakietowi `communication`, klient (`Controller`) i serwer (`Server`) nie komunikują się za pomocą “gołych” socketów i bajtów. Zawiera dwa podpakiety: `messages` i `streams`.

2.2.1 Messages

Podstawą komunikacji są wiadomości. Komunikacja odbywa się jednocześnie na dwa sposoby: request-response (`xxRequest-xxResponse`), gdzie klient pyta o coś serwer, a serwer odpowiada, oraz poprzez wiadomości `xxNotice`, które wysyła serwer do klientów, informując ich o jakiejś zmianie.

Całość stanu gry jest jawna dla każdego klienta. Przesyłane są zmiany stanu, chociaż jest możliwość poproszenia o cały stan.

Wiadomości są silnie otypowane jako `ClientToServerMessage` lub `ServerToClientMessage`.

2.2.2 Streams

Streamy obudowują komunikację, tak aby można było przysyłać wyłącznie wiadomości z kontrolą typów oraz umożliwiając abstrakcję na komunikację przez socket albo lokalnie.

Inne pakiety powinny korzystać *wyłącznie* z odpowiednio `ClientMessageStream` lub `ServerMessageStream` opakowujących inne rzeczy, np.:

```
new ClientMessageStream(new TCPMessageStream(new Socket(...)))
```

`{Client, Server}MessageStream` może opakowywać `TCPMessageStream` opakowujący `Socket` albo `PipedMessageStream` opakowujący lokalny pipe. Chociaż oszczędnie `PipedMessageStream` nie jest używany, był przydatny do testów.

Dziwna nieco architektura jest konsekwencją chęci zachowania kontroli typów wiadomości.

2.3 Server

Serwer jest zbudowany jako samowystarczalny. `Controller` musi zrobić:

```
Server server = new Server();
Thread serverThread = server.runInNewThread();
// ...
serverThread.join();
```

I nie musi się serwerem więcej przejmować.

Serwer ma trzy główne części:

- `acceptorTh` — wątek nasłuchujący na `ServerSocket`, akceptujący każdego klienta i dodającego go do listy klientów z utworzeniem właściwych mu wątków/struktur danych.

- klienci — każdy klient ma dwa wątki: jeden pisze na `ServerMessageStream`, drugi zeń czyta i wrzuca wiadomości na kolejkę wydarzeń. W razie przepełnienia buforów, klient jest odłączany.
- `mainLoopTh` — sekwencyjnie przetwarza kolejne rządania/wydarzenia z kolejki i wrzuca odpowiedzi na bufor klientów. Przechowuje stan gry.

Testy serwera to nie testy jednostkowe, a małe programy, które pomogły przy debugowaniu.

3 Napotkane trudności i uwagi do architektury

3.1 Napotkane trudności

Jedynym naprawdę trudnym bugiem był problem z serializacją obiektu `Game`. Obiekt `Player` rozszerzał `Hand` i implementował `Serializable`, ale `Hand` nie implementowało `Serializable`, przez co nie przysyłał się stan ręki.

3.2 Przemyślenia dot. architektury

Architektura z pewnością nie jest doskonała. Część decyzji była pewnie błędna, albo przynajmniej nieoptymalna. Niemniej, równie ważne co polepszanie architektury, jest to, aby wiedzieć, kiedy przestać. Obecny kod spełnia założenia projektu i opiera się na poprzednich decyzjach, a refactoring kosztuje.