Sprawozdanie - projekt z programowania w języku Java

Grzegorz Bujak Arkadiusz Markowski grupa 2ID11B
Politechnika świętokrzyska w Kielcach

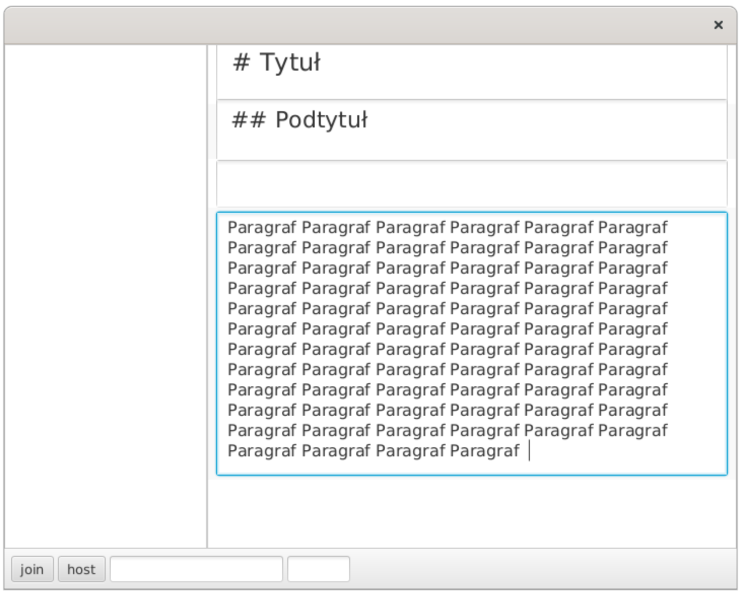
Spis treści

Założenia projektu

Nasz projekt polegał na napisaniu edytora tekstu w języku Java z wykorzystaniem biblioteki JavaFX. Edytor ma funkcję pisania tego samego dokumentu przez wiele osób w tym samym czasie. Podczas pisania, synchronizuje dokument pomiędzy użytkownikami.

Edytor tekstu traktuje dokument jako listę paragrafów (bloków tekstu). Gdy użytkownik wciśnie klawisz enter, tworzony jest nowy blok pod obecnie edytowanym. Dzięki takiemu rozwiązaniu możemy używać widżetu `ListView` z JavaFX do wyświetlania dokumentów.

Widżet ten pozwala na rysowanie na ekranie tylko tylu pól tekstowych, ile jest w tej chwili widoczne. Jedno duże pole tekstowe na cały dokument mogłoby powodować problemy z płynnym przewijaniem strony przy ogromnych dokumentach.



Rysunek 1: interfejs aplikacji

Moduł sieciowy edytora używa topologii klient-serwer, ale serwer nie jest aplikacją terminalową. Interfejs graficzny zapewnia możliwość dołączenia do serwera oraz hostowania własnego dokumentu.

Moduł sieciowy

Moduł sieciowy naszego projektu składa się z klas:

- `Networker<T extends Serializable>`

  Jedna instancja dla serwera i klienta. Jeśli program działa w trybie serwera, tworzy instancję `ServerSocket`, która ciągle akceptuje nowych klientów. Dla każdego klienta tworzy nową instancję `SocketHandler<T>`, która będzie odbierała wiadomości. `Dispatcher<T>` dostaje każdego podłączonego klienta.

  W trybie klienta, tworzy `Dispatcher<T>`, który będzie wysyłał wiadomości tylko do serwera oraz `SocketHandler<T>`, który będzie odbierał wiadomości od serwera.

- `Dispatcher<T extends Serializable>`

  Klasa odpowiada za wysyłanie wiadomości.

  Przechowuje listę wszystkich wiadomości (obiekty klasy `T`) oraz `HashMap<Socket, Integer>`, który przechowuje sockety, do których ma wysyłać wiadomości oraz `Integer` będący ID ostatniej wiadomości

wysłanej do socketa. Z mapy korzysta metoda `void dispatch()`, która sprawdza ilość wiadomości i iteruje po socketach wysyłając im po kolei wszystkie wiadomości, których nie dostali.

Zapewnia medotę `void addAndDispatch(T)`, która dodaje wiadomość typu `T` do listy wiadomości i wysyła tą wiadomość do wszystkich socketów.

Ma też metodę `void addSocket(Socket)`, która dodaje Socket do mapy i wywołuje metodę `dispatch()`, która wyśle temu socketowi wszystkie wiadomości dodane do Dispatchera od początku działania programu.

- `SocketHandler<T extends Serializable>`

  Klasa jest odpowiedzialna za odbieranie wiadomości. Do jej konstruktora należy podać domknięcie (obiekt interfejsu funkcjonalnego `Consumer<T>`). Obiekt w nieskończoność odbiera nadchodzące wiadomości i wywołuje domknięcie z każdą wiadomością.

Wszystkie klasy powinny działać na własnych wątkach.

Edytor tekstu

Edytor tekstu to lista `ListView` bloków tekstowych.

Bloki tekstowe to obiekty naszej własnej klasy `ExpandingTextArea`. Jest to `TextArea`, która zmienia swoją wysokość podczas pisania.

Wciśnięcie klawicza enter nie powoduje wstawienia nowej linii w polu tekstowym, lecz jest przechwytywane przez klasę `MainGuiController`, która wstawia do listy nowe pole tekstowe po tym, w którym znajduje się kursor oraz kopiuje do niego wszystko co znajdowało się za kursorem w polu tekstowym.

Wciśnięcie backspace, gdy kursor znajduje się na początku pola tekstowego powoduje skasowanie tego pola tekstowego i skopiowanie jego zawartości za kursorem do poprzedniego pola tekstowego.

Rozsyłane wiadomości

- pochodne klasy `DocumentAction`.
  - `AddBlockAction` - dodanie nowego bloku w edytorze.
  - `ChangeBlockAction` - zmiana tekstu w bloku.
  - `RemoveBlockAction` - usunięcie bloku.

  Pochodne tej klasy mają metody:

  - `void commit(Document)` - do wprowadzania zmian w edytowanym w tej chwili dokumencie.
  - `void commit(List<SerializableBlock>)` - do wprowadzania zmian w dokumencie znajdującym się w postaci serializowanej w `Datastore`.

Produkowanie wiadomości przez edytor

Podczas pisania, edytor generuje wiadomości. Są do obiekty klasy `DocumentAction`. Tutaj opisaliśmy warunki generowania wiadomości:

- zmiana zawartości pola tekstowego:

  -> wygenerowanie wiadomości `ChangeBlockAction`.

- usunięcie bloku:

  -> wygenerowanie wiadomości `RemoveBlockAction`,

-> wygenerowanie wiadomości `ChangeBlockAction` dla bloku przed usuniętym na wypadek przeniesienia do niego tekstu z usuniętego.

- dodanie bloku:

  -> wygenerowanie wiadomości `AddBlockAction`,

  -> wygenerowanie wiadomości `ChangeBlockAction` dla bloku, w którym znajdował się kusor na wypadek przeniesienia z niego tekstu do nowego bloku.

Nie jest możliwe przeniesienie tekstu do następnego bloku, dlatego żadna akcja nie generuje wiadomości `ChangeBlockAction` dla bloku znajdującego się pod aktywnym (`AddBlockAction` przewiduje to, że nowo stworzony blok będzie miał jakąś początkową zawartość).

Reprezentacja dokumentu

Obiekt klasy `Document` reprezentuje edytowany w tej chwili dokument. Klasa zawiera pole typu `ObservableList<Block>`, który nie może być serializowany.

Funkcjonalność programu w dużym stopniu zależy od tego obiektu. Jest on obserwowany dwukierunkowo przez `MainGuiController`. Dzięki temu, każda zmiana w polach tekstowych jest automatycznie synchronizowana z obiektami klasy `Block` przechowywanymi w obiekcie `Document`.

`Block` jest klasą abstrakcyjną. Jedyną klasą dziedziczącą po `Block` jest `TextBlock`, który reprezentuje blok tekstu. Nie zdążyliśmy zaimplementować innych rodzajów bloków, które mogłyby reprezentować np. wklejone zdjęcie.

Serializowanie dokumentów

Przez brak możliwości serializacji bloków oraz obiektów klasy `Document`, stworzyliśmy serializowalne odpowiedniki do przechowywania w obiekcie `Datastore` i do zapisywania na dysku:

- `Block` -> `SerializableBlock`

  Klasa abstrakcyjna posiadająca abstrakcyjne metody:

  - `abstract Block toBlock()`:

    Konwersja na obiekt `Block` do populacji obiektu `Document`.

  - `abstract void setContent(String newContent)`:

    Zmiana zawartości. Może to być nowy tekst lub nowa ścieżka do zdjęcia. Tej funkcji używają obiekty klasy `DocumentAction`.

- `TextBlock` -> `SerializableTextBlock`:

  Klasa nieabstrakcyjna reprezenująca blok tekstu. Zawarta w całości w obiekcie `ChangeBlockAction`.

Wnioski

Wykonując projekt, utrwaliliśmy wiedzę na temat programowania obiektowego oraz przećwiczyliśmy pracę w grupie z wykorzystaniem programu Git.