Sprawozdanie - projekt z programowania w języku Java

Grzegorz Bujak Arkadiusz Markowski grupa 2ID11B
Politechnika świętokrzyska w Kielcach

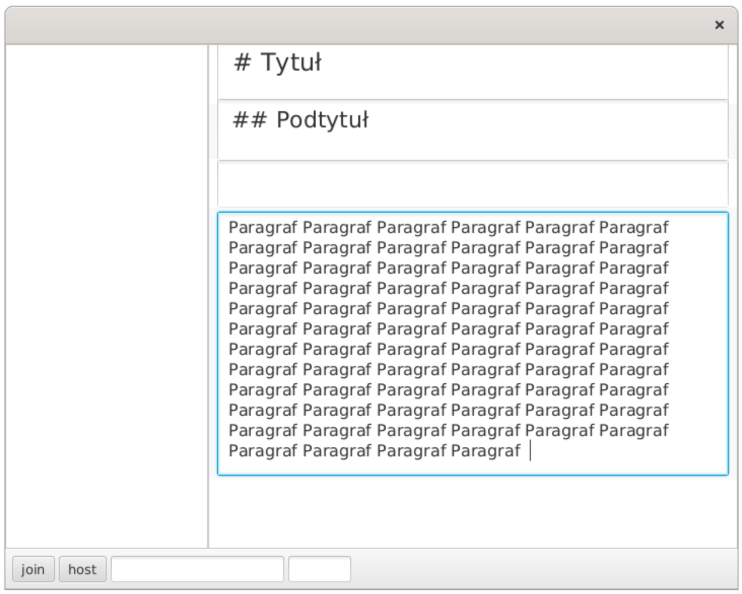
Spis treści

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.

Networker<T extends Serializable>

```java
package lan_editor.networking;

// import ...

/**
 * Klasa zajmująca się komunikacją sieciową z serwerem i innymi klientami
 */

public class Networker<T extends Serializable> implements Runnable {
    private Dispatcher<T> dispatcher;
    private boolean iAmServer;

    private int port;
    private String address;

    private Consumer<T> consumer;

    private TypeToken<T> typeToken;

    public Networker(
            boolean isServer, String address, int port,
            Consumer<T> onReceive, TypeToken<T> typeToken) {
        if (address == null && !isServer)
            throw new IllegalArgumentException("adres nie moze byc null dla klienta");
        this.address = address;
        this.port = port;
        this.iAmServer = isServer;
        this.consumer = onReceive;
        this.dispatcher = new Dispatcher<T>();
        this.typeToken = typeToken;
    }

    public static <T extends Serializable> Networker<T> makeServer(
        int port, Consumer<T> onReceive, TypeToken<T> responseTypeToken) {
        return new Networker<T>(true, null, port, onReceive, responseTypeToken);
    }
```

```java
public static <T extends Serializable> Networker<T> makeClient(
    String address, int port, Consumer<T> onReceive, TypeToken<T> responseTypeToken) {
    return new Networker<T>(false, address, port, onReceive, responseTypeToken);
}

ServerSocket sock;
@Override
public void run() {
    if (iAmServer)
        server();
    else client();
}

private void server() {
    try {
        sock = new ServerSocket(port);
    } catch (IOException e) {
        e.printStackTrace();
    }

    while (true) {
        try {
            var newClient = sock.accept();
            System.out.println("accepted: " + newClient.getInetAddress().getHostName());
            dispatcher.addSocket(newClient);
            var thread = new Thread(new SocketHandler<T>(
                consumer, dispatcher, newClient, typeToken));
            thread.setDaemon(true);
            thread.start();
        } catch (IOException e) {e.printStackTrace();}
    }
}

private void client() {
    dispatcher = new Dispatcher<T>();
    Socket sock;
    try {
        sock = new Socket();
        sock.connect(new InetSocketAddress(address, port));
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(1);
        return;
    }
    dispatcher.addSocket(sock);
    var handlerThread = new Thread(new SocketHandler<T>(
        consumer, dispatcher, sock, typeToken));
    handlerThread.setDaemon(true);
    handlerThread.start();
}

public synchronized void send(T item) {
    dispatcher.addAndDispatch(item);
}
```

```
}
```

Dispatcher<T extends Serializable>

```
package lan_editor.networking;

// import ...

/**
 * Dispatcher rozsyła T do socketów, które przechowuje
 * wszystkie metody są synchronizowane
 *
 * @param <T> to rozsyłany obiekt implementujący Serializable
 */
public class Dispatcher<T extends Serializable> {
    ArrayList<T> items = new ArrayList<>();

    /// Przechowuje socket oraz id ostatniego itemu, który został wysłany do tego socketa
    HashMap<Socket, Integer> sockets = new HashMap<>();

    public synchronized void addSocket(Socket sock) {
        System.out.println("added sock: " + sock.getInetAddress().getHostName());
        sockets.put(sock, 0);
        dispatch();
    }

    public synchronized void remove(Socket sock) {
        sockets.remove(sock);
    }

    public synchronized void addAndDispatch(T item) {
        items.add(item);
        dispatch();
    }

    public synchronized void dispatch() {
        for (var entry : sockets.entrySet()) {
            int lastClientAction = entry.getValue();
            Socket clientSocket  = entry.getKey();
            if (items.size() > lastClientAction) {
                for (int i = lastClientAction; i < items.size(); i++) {
                    var item = items.get(i);
                    try {
                        var writer = new PrintWriter(clientSocket.getOutputStream(), true);
                        var gson = new Gson();
                        var json = gson.toJson(item);
                        System.out.println("sending " + json + " to "
                            + clientSocket.getInetAddress().getHostName());
                        writer.println(json);
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
```

```
                entry.setValue(items.size());
            }
        }
    }
}
```

SocketHandler<T extends Serializable>

```
package lan_editor.networking;

// import ...

public class SocketHandler<T extends Serializable> implements Runnable {
    private BufferedReader reader;
    private Socket sock;

    private Consumer<T> consumer;
    private Dispatcher<T> dispatcher;

    private TypeToken<T> typeToken;

    public SocketHandler(
            Consumer<T> onReceive, Dispatcher<T> dispatcher,
            Socket sock, TypeToken<T> typeToken) {
        this.typeToken = typeToken;
        this.consumer = onReceive;
        this.sock = sock;
        this.dispatcher = dispatcher;
        try {
            reader = new BufferedReader(new InputStreamReader(sock.getInputStream()));
        } catch (IOException e) {e.printStackTrace();}
    }

    @Override
    public void run() {
        while (!sock.isClosed()) {
            T received;

            try {
                received = new Gson().fromJson(
                    reader.readLine(), typeToken.getType());
            } catch (Exception e) {
                e.printStackTrace();
                dispatcher.remove(sock);
                return;
            }

            // uruchom na wątku interfejsu graficznego
            Platform.runLater(() -> consumer.accept(received));
        }
        dispatcher.remove(sock);
    }
}
```

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.

`ExpandingTextArea`

```
package lan_editor.gui.widgets;

// import ...

public class ExpandingTextArea extends TextArea {
    private TextBlock textBlock;
    private Text content;
    private double currentHeight = getFont().getSize() + 20;

    public TextBlock getTextBlock() {
        return textBlock;
    }

    public ExpandingTextArea(TextBlock textBlock) {
        super();
        wrapTextProperty().setValue(true);
        setManaged(true);
        setPrefWidth(500);

        this.textBlock = textBlock;
        this.content = this.textBlock.getContent();
        this.textProperty().bindBidirectional(this.content.textProperty());

        detectFontChange();
        fitToText();

        this.setOnKeyPressed(keyEvent -> {
            detectFontChange();
            fitToText();
        });
    }

    public void fitToText() {
        if (currentHeight < 50) currentHeight = 50;
        Text tx = (Text) this.lookup(".text");
        if (tx != null) {
            currentHeight = Math.max(
```

```
                    getFont().getSize() + 20,
                    tx.getBoundsInLocal().getHeight() + 20);
        }
        setPrefHeight(currentHeight);
    }

    private void detectFontChange() {
        if (content.getText().startsWith("# "))
            this.setFont(TextTypes.header1);
        else if (content.getText().startsWith("## "))
            this.setFont(TextTypes.header2);
        else if (content.getText().startsWith("### "))
            this.setFont(TextTypes.header3);
        else if (this.getFont() != TextTypes.defaultFont)
            this.setFont(TextTypes.defaultFont);
        else if (this.getText().startsWith("```"))
            this.setFont(TextTypes.monoFont);
    }
}
```

Formatowanie tekstu

Edytor zapewnia formatowanie tekstu podobne jak w formacie markdown. Gdy blok tekstu zaczyna się od specjalnego ciągu znaków, zmieniana jest czcionka całego bloku.

```
package lan_editor.gui.constants;

import javafx.scene.text.Font;

public class TextTypes {
    public static final Font header1 =
            new Font(24); // #
    public static final Font header2 =
            new Font(22); // ##
    public static final Font header3 =
            new Font(20); // ###
    public static final Font defaultFont =
            new Font(16); // ```
    public static final Font monoFont =
            new Font("monospace", 16);
}
```

`MainGuiController`

```
package lan_editor.gui;

// import ...

/**
 * Kontroler głównego okna
 */

public class MainGuiController {
    private Networker<Action> networker;
```

```
private Datastore datastore = new Datastore();

private Document document = null;

public void setDocument(Document doc) {
    document = doc;
    mainListView.setItems(document.getBlocks());
}
public Document getDocument() {
    return document;
}

@FXML
private TreeView<?> mainTreeView;
@FXML
private ListView<Block> mainListView;
@FXML
private ToolBar mainToolBar;
@FXML
private Button joinButton;
@FXML
private Button hostButton;
@FXML
private TextField addressField;
@FXML
private TextField portField;

@FXML
public void initialize() {
    mainListView.setCellFactory(
            blockListView -> new BlockListCell());
    mainListView.addEventFilter(KeyEvent.KEY_PRESSED, this::handleKeyEvent);

    joinButton.setOnAction(this::handleJoin);
    hostButton.setOnAction(this::handleHost);

    var doc = new Document();
    doc.getBlocks().add(new TextBlock(""));
    this.setDocument(doc);
}

private Consumer<Action> consumer = action -> {
    action.getDocumentAction().commit(this.getDocument());
    var node = this.getDocument().getBlocks().get(action.getBlockIndex()).getNode();
    if (node instanceof ExpandingTextArea) ((ExpandingTextArea) node).fitToText();
};

private void handleJoin(ActionEvent ev) {
    var url = addressField.getText();
    int port = 0;
    try {
        port = Integer.parseInt(portField.getText());
    } catch (NumberFormatException e) {
```

```
            portField.setText("");
            return;
        }

        this.networker = Networker.makeClient(
                url, port,
                consumer,
                new TypeToken<Action>(){}
        );

        var thread = new Thread(this.networker);
        thread.setDaemon(true);
        thread.start();

        portField.setDisable(true);
        addressField.setDisable(true);
    }

    private void handleHost(ActionEvent ev) {
        int port = 0;
        try {
            port = Integer.parseInt(portField.getText());
        } catch (NumberFormatException e) {
            portField.setText("");
            return;
        }

        this.networker = Networker.makeServer(
                port,
                action -> { // Jeśli jesteś serwerem, roześlij otrzymaną akcję
                    this.consumer.accept(action);
                    this.networker.send(action);
                },
                new TypeToken<Action>(){}
        );

        var thread = new Thread(this.networker);
        thread.setDaemon(true);
        thread.start();

        portField.setDisable(true);
        addressField.setDisable(true);
    }

    private void handleKeyEvent(KeyEvent ev) {
        // Enter tworzy nowe pole
        var selected = mainListView.getScene().getFocusOwner();
        if (ev.getCode() == KeyCode.ENTER) {
            if (selected instanceof ExpandingTextArea && !ev.isShiftDown()) {
                var selTextArea = (ExpandingTextArea) selected;
                var caret = selTextArea.getCaretPosition();
                var newBlock = new TextBlock(selTextArea.getText().substring(caret));
                selTextArea.setText(selTextArea.getText(0, caret));
                var index = document.getBlocks().indexOf(selTextArea.getTextBlock());
```

```
        document.getBlocks().add(
                index + 1,
                newBlock
        );
        newBlock.getNode().requestFocus();
        ev.consume();

        if (this.networker != null) {
            networker.send(new Action(
                new ChangeBlockAction("", index, selTextArea.getText())));
            networker.send(new Action(new AddBlockAction(
                    "", index + 1, newBlock.getContent().getText()
            )));
        }
    }
}

// Backspace usuwa puste pole
if (ev.getCode() == KeyCode.BACK_SPACE) {
    if (selected instanceof ExpandingTextArea && !ev.isShiftDown()) {
        var selTextArea = (ExpandingTextArea) selected;
        var caretPos = selTextArea.getCaretPosition();
        if (caretPos != 0) return;

        var remainingText = selTextArea.getText();
        var index = document.getBlocks().indexOf(selTextArea.getTextBlock());
        if (index == 0) return;

        var prevBlock = document.getBlocks().get(index - 1);
        if (prevBlock instanceof TextBlock) {
            var prevText = (TextBlock) prevBlock;
            var newCaret = prevText.getContent().getText().length();
            prevText.getContent().setText(
                    prevText.getContent().getText().concat(remainingText)
            );
            document.getBlocks().remove(index);
            prevText.getNode().positionCaret(newCaret);
            prevText.getNode().requestFocus();
        }

        if (this.networker != null) {
            networker.send(new Action(new RemoveBlockAction("", index - 1)));
            if (prevBlock instanceof TextBlock) {
                networker.send(new Action(new ChangeBlockAction(
                        "", index - 1, ((TextBlock) prevBlock).getContent().getText()
                )));
            }
        }
    }
}

if (selected instanceof ExpandingTextArea && this.networker != null) {
    var textArea = (ExpandingTextArea) selected;
    networker.send(new Action(
```

```
                    new ChangeBlockAction(
                        "",
                        document.getBlocks().indexOf(textArea.getTextBlock()),
                        textArea.getText()
            )));
        }
    }
}
```

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 używją 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.