Minigry

Karol Marciniak U-15991

Przemysław Kuras U-20916

Szymon Monasterski U-20096

Adam Orłowski U-20103

1. Cel i opis projektu

a. Celem projektu było zaprojektowanie i implementacja aplikacji komponentowej, umożliwiającej integrację i uruchamianie mini gier stworzonych jako odrębne komponenty.
b. W skład projektu wchodzą cztery mini gry: Snake, Saper i TicTacToe, a także komponent tła — AnimatedBackground. Wszystkie komponenty są osadzone w aplikacji nadrzędnej, która odpowiada za ich inicjalizację, obsługę i przełączanie.

2. Wymagania funkcjonalne

a. Snake:
    i. Włączenie gry10
    ii. Zliczanie punktów
    iii. Zapis wyniku
    iv. Serializacja danych
b. Saper:
    i. Włączenie gry
    ii. Zliczanie punktów
    iii. Zapis wyniku
    iv. Serializacja danych
c. TicTacToe:
    i. Włączenie gry
    ii. Zliczanie punktów
    iii. Zapis wyniku
    iv. Serializacja danych
d. AnimatedBackground:
    i. Włączanie animacji
e. Menu:
    i. Wybór gry
    ii. Wybór użytkownika
    iii. Wybór poziomu
    iv. Serializacja

3. Komponenty

a. Snake

i. Opis:
Klasyczna gra zręcznościowa, w której gracz steruje wężem poruszającym się po planszy. Celem gry jest zbieranie jedzenia, które powoduje wydłużenie węża, bez zderzenia się ze ścianami lub samym sobą.
ii. Zasada działania:
Gracz steruje wężem za pomocą klawiatury (strzałki). Na planszy losowo pojawiają się punkty (jedzenie), które wąż zbiera. Każde zjedzenie punktu wydłuża węża i zwiększa wynik gracza. Gra kończy się, gdy wąż uderzy w ścianę lub własne ciało. Czas gry jest ograniczony.

b. Saper

i. Opis:
Gra logiczna polegająca na odkrywaniu pól planszy bez trafienia na minę.

Gracz musi na podstawie odkrytych liczb wnioskować, gdzie znajdują się miny.

ii. Zasada działania:

Plansza składa się z pól, z których część ukrywa miny. Kliknięcie na pole może odkryć liczbę (oznaczającą liczbę sąsiadujących min), puste pole (odkrywa większy obszar) lub minę (gra kończy się porażką). Celem jest odkrycie wszystkich pól niebędących minami. Gracz może oznaczać pola flagami. Poziom trudności wpływa na rozmiar planszy i liczbę min.

c. TicTacToe

i. Opis:

Prosta gra logiczna dla dwóch graczy, których celem jest ustawienie trzech symboli w jednej linii na planszy 3x3.

ii. Zasada działania:

Gracze na zmianę stawiają swoje symbole (kółko i krzyżyk) na planszy. Wygrywa ten, kto pierwszy ułoży trzy swoje symbole w poziomie, pionie lub na skos. Gra może zakończyć się remisem. W aplikacji przeciwnikiem może być drugi gracz lokalny lub prosty bot (jeśli zaimplementowano).

d. AnimatedBackground

i. Opis:

Komponent estetyczny, odpowiadający za wyświetlanie animowanego tła w aplikacji: poruszający się wąż, wybuchające bomby oraz kółka i krzyżyki

4. Przypadki testowe

a. Każdy komponent posiada przynajmniej jeden przypadek testowy jednostkowy oraz jeden przypadek testowy automatyczny, sprawdzający jego podstawową funkcjonalność.

b. Jednostkowy:

Numer testu	Nazwa testu	Opis	Kroki testowe	Oczekiwany wynik
T-001	Konstruktor SnakeComp()	Sprawdzenie poprawnego działania konstruktora	wywołanie konstruktora	Pojawiło się okno
T-002	Konstruktor SaperComp()	Sprawdzenie reakcji po kliknięciu w pole z miną	włączenie gry, wybieranie pola dopóki nie trafimy na minę	Gra zakończyła się z oknem przegranego
T-003	Konstruktor TicTacToe,Comp()	Sprawdzenie warunku zwycięstwa przy trzech znakach w linii	włączenie gry, wygranie partii	Gra zakończyła się z oknem zwycięzcy
T-004	Konstruktor AnimatedBackgroundComp()	Sprawdzenie, czy komponent się renderuje bez błędów	włączenie menu, sprawdzenie czy animacja działa	Wąż porusza się, kółka i krzyżyk pojawiają się i znikają

c. Automatyczny

Numer testu	Nazwa testu	Opis	Kroki testowe	Oczekiwany wynik
T-001	Inkrementacja węża	Sprawdzenie poprawnego zwiększania długości po zjedzeniu jabłka	włączenie gry, zjedzenie jabłka	Wąż zwiększył się o jeden segment
T-002	Przegranie sapera	Sprawdzenie reakcji po kliknięciu w pole z miną	włączenie gry, wybieranie pola dopóki nie trafimy na minę	Gra zakończyła się z oknem przegranego
T-003	Wygranie TicTacToe	Sprawdzenie warunku zwycięstwa przy trzech znakach w linii	włączenie gry, wygranie partii	Gra zakończyła się z oknem zwycięzcy
T-004	Test animacji	Sprawdzenie, czy komponent się renderuje bez błędów	włączenie menu, sprawdzenie czy animacja działa	Wąż porusza się, kółka i krzyżyk pojawiają się i znikają

5. Przypadki wywołań

a. Komponent AnimatedBackground:
    i. podczas tworzenia okna menu - otwieranie aplikacji, przycisk Back w Settings i Scores
    ii. podczas tworzenia okna Settings - przycisk Settings
    iii. podczas tworzenia okna Scores - przycisk Scores
b. Komponent Snake:
    i. podczas tworzenia okna Snake - przycisk Snake
c. Komponent Saper:
    i. podczas tworzenia okna Saper - przycisk Saper
d. Komponent TicTacToe:
    i. podczas tworzenia okna TicTacToe - przycisk TicTacToe

6. Zbiór cech komponentów

a. AnimatedBackground, Snake, Saper, TicTacToe:
    i. pola prywatne
    ii. publiczne gettery i settery do pól prywatnych
    iii. publiczne funkcje do sterowania komponentem
    iv. konstruktor bezargumentowy
    v. autonomiczny moduł aplikacji złożony z klas współpracujących w ramach wspólnej funkcji, posiadający własną logikę, dane oraz interfejs komunikacji z otoczeniem.
    vi. serializacja

7. Programowanie komponentowe versus obiektowe

a. Chociaż programowanie obiektowe (OOP) jest powszechnie stosowane przy tworzeniu aplikacji, w naszym projekcie skupiono się na programowaniu komponentowym (CBSE – Component-Based Software Engineering). Komponentowe podejście pozwala wydzielić niezależne moduły aplikacji, z których każdy realizuje określoną funkcję, posiada własną logikę, interfejs oraz dane.
b. Każdy z komponentów – Snake, Saper, TicTacToe, AnimatedBackground – został zaprojektowany jako samodzielna jednostka funkcjonalna, gotowa do wielokrotnego użycia i łatwego testowania. Choć komponenty wewnętrznie wykorzystują elementy programowania obiektowego (np. klasy z prywatnymi polami, publicznymi metodami i serializacją), główną ideą architektoniczną projektu było rozdzielenie logiki na niezależne, komunikujące się komponenty

8. Dokumentacja użytkownika komponentów

a. AnimatedBackground
   i. Typ: komponent wizualny
   ii. Pakiet: `org.example.animatedbackgroundcomp`
   iii. Zależności: JavaFX (`javafx.scene.*`, `javafx.animation.*`, `javafx.util.Duration`, `javafx.media.*`)
   iv. Wymagania:
      1. JDK: minimum Java 17
      2. Framework: JavaFX (musi być skonfigurowany w projekcie)
      3. Media: plik dźwiękowy określony w konfiguracji (`AnimationConfig.getMusicFile()`)
   v. Uruchomienie komponentu:
   vi. Główne metody API:

```java
public class AnimationConfig {
    public AnimationConfig() {
    }

    public AnimationConfig(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getWidth() { return this.width; }

    public void setWidth(int width) { this.width = width; }

    public int getHeight() { return this.height; }

    public void setHeight(int height) { this.height = height; }

    public boolean isMusicEnabled() { return this.isMusicEnabled; }

    public void setMusicEnabled(boolean musicEnabled) { this.isMusicEnabled = musicEnabled; }

    public String getMusicFile() { return this.musicFile; }

    public void setMusicFile(String musicFile) { this.musicFile = musicFile; }

    public int getVolMusic() { return this.volMusic; }

    public void setVolMusic(int volMusic) {
        if (volMusic < 0) {
            this.volMusic = 0;
        } else if (volMusic > 100) {
            this.volMusic = 100;
        } else {
            this.volMusic = volMusic;
        }

    }
```

```java
public boolean isCrossVisible() { return this.isCrossVisible; }

public void setCrossVisible(boolean crossVisible) { this.isCrossVisible = crossVisible; }

public int getCrossSize() { return this.crossSize; }

public void setCrossSize(int crossSize) { this.crossSize = crossSize; }

public String getCrossColor() { return this.crossColor; }

public void setCrossColor(String crossColor) { this.crossColor = crossColor; }

public boolean isCircleVisible() { return this.isCircleVisible; }

public void setCircleVisible(boolean circleVisible) { this.isCircleVisible = circleVisible; }

public int getCircleSize() { return this.circleSize; }

public void setCircleSize(int circleSize) { this.circleSize = circleSize; }

public String getCircleColor() { return this.circleColor; }

public void setCircleColor(String circleColor) { this.circleColor = circleColor; }

public boolean isSnakeVisible() { return this.isSnakeVisible; }

public void setSnakeVisible(boolean snakeVisible) { this.isSnakeVisible = snakeVisible; }

public String getSnakeColor() { return this.snakeColor; }

public void setSnakeColor(String snakeColor) { this.snakeColor = snakeColor; }

public int getSnakeLength() { return this.snakeLength; }

public void setSnakeLength(int snakeLength) { this.snakeLength = snakeLength; }
public boolean isBackgroundVisible() { return this.isBackgroundVisible; }

public void setBackgroundVisible(boolean backgroundVisible) { this.isBackgroundVisible = backgroundVisible; }

public String getBackgroundColor() { return this.backgroundColor; }

public void setBackgroundColor(String backgroundColor) { this.backgroundColor = backgroundColor; }

public boolean isLineVisible() { return this.isLineVisible; }

public void setLineVisible(boolean lineVisible) { this.isLineVisible = lineVisible; }

public String getLineColor() { return this.lineColor; }

public void setLineColor(String lineColor) { this.lineColor = lineColor; }

public boolean isExplosionEnabled() { return this.isExplosionEnabled; }

public void setExplosionEnabled(boolean explosionEnabled) { this.isExplosionEnabled = explosionEnabled; }

public int getMinExploDelay() { return this.minExploDelay; }

public void setMinExploDelay(int minExploDelay) { this.minExploDelay = minExploDelay; }

public int getMaxExploDelay() { return this.maxExploDelay; }

public void setMaxExploDelay(int maxExploDelay) { this.maxExploDelay = maxExploDelay; }
```

```
public void saveSettings(String filePath)
public String readSetting(String param1, String param2)
public void loadSettings(String filePath)
public void init()
public static void playExplosion(Node node, Pane parentPane)
public void dispose()
public void updateSettings()
public void setAnimatedSnakeColor(String color)
public AnimationConfig getConfig() { return this.config; }

public void setConfig(AnimationConfig config) {
```

9. Repozytorium GitHub

a. https://github.com/Feste790/Minigames/tree/main/ZPK_Lato2025_Marciniak_Or%C5%82owski_Monasterski_Kuras

10. Użyte narzędzia programistyczne

a. IntelliJ IDEA / Eclipse - środowisko programistyczne (IDE) używane do tworzenia i uruchamiania aplikacji.
b. Maven - system budowania i zarządzania zależnościami projektu.
c. JavaFX - biblioteka graficzna do budowy interfejsu użytkownika i animacji.
d. GitHub - system kontroli wersji wykorzystywany do zarządzania kodem źródłowym.

11. Materiały źródłowe

a. Dokumentacja JavaFX (https://openjfx.io)
b. Dokumentacja Maven (https://maven.apache.org)
c. Dokumentacja języka Java (https://docs.oracle.com/en/java/)

12. Wnioski

Projekt aplikacji komponentowej umożliwił praktyczne zastosowanie zasad programowania komponentowego, w odróżnieniu od tradycyjnego podejścia obiektowego. Dzięki podziałowi na niezależne komponenty (takie jak Snake, Saper, TicTacToe, AnimatedBackground) możliwe było tworzenie modułów autonomicznych, które posiadają własną logikę, interfejs oraz dane.
Wszystkie komponenty mogą być uruchamiane i rozwijane niezależnie, co ułatwia ich testowanie, ponowne użycie i modyfikację.
Projekt pozwolił również na utrwalenie umiejętności z zakresu:

- tworzenia GUI w JavaFX,
- organizowania kodu w sposób modułowy,
- konfiguracji projektu przy użyciu Mavena,
- obsługi zdarzeń i animacji.

Z perspektywy deweloperskiej, podejście komponentowe pozwala lepiej zarządzać złożonością większych aplikacji i ułatwia pracę zespołową dzięki wyraźnemu podziałowi odpowiedzialności.