Politechnika Krakowska

Wydział Fizyki, Matematyki i Informatyki

ZADANIE 6

Zaawansowane techniki programowania

Prowadzący: dr inż. Jerzy Jaworowski

II stopień - rok 1, semestr 1

wykonanie:

Piotr Adam Tomaszewski

Nr albumu: 104896

Wstęp

Celem zadania było zaimplementowanie programu do obliczania pola powierzchni obszarów. Na samym początku należało utworzyć ‘połączenie’ z metodą IGameMonitor a następnie zarejestrowanie się przez wywołanie metody register. Jeśli przebiegła ona pomyślnie, to należy pobrać dane z bazy danych. W celu zainicjowania programu servlet przekazuje dane do komponentu (metoda general udostępniona interfejsem IPlaneRemote komponentu PlaneImpl). Program wylicza kolejne pola obszarów zależnych od współrzędnej z i zwraca maksymalną wartość pola z tych obszarów.

Organizacja plików

IGameRemote.java

Interfejs ***IGameRemote*** posiadający deklarację metod, która pozwalają na zarejestrowanie użytkownika w systemie.

Deklaracja metody ***register*** rejestrująca użytkownika w systemie - zwraca true jeżeli proces rejestracji zakończył się poprawnie. Jeżeli rejestracja zakończyła się niepowodzeniem, metoda register zwraca wartość false.

@param hwork - numer zadania

@param album – numer albumu studenta

@return wartość logiczna czy połączenie przebiegło pomyślnie

```
public boolean register(int hwork, String album);
```

Rysunek 1 Deklaracja metody register

IPlaneRemote.java

Interfejs ***IPlaneRemote*** zawierający deklaracje metod wykorzystywanych w klasie PlaneImpl niezbędny do użycia bean'a przekazującego listę punktów z servletu do klasy PlaneImpl.

Deklaracja metody ***general*** wywoływana przez bean z serwletu określająca wykonywanie kolejnych metod w klasie.

@param arr - lista wszystkich pobranych punktów z bazy danych

@return - maksymalne pole przestrzeni

```
public double general (List<Float> arr);
```

Rysunek 2 Deklaracja metody general

Deklaracja metody ***prepare*** pozwalającej na przyporządkowanie określonych wartości pobranych z bazy do listy jako współrzędne punktów w przestrzeni 3D. Jeśli kolejna grupa punktów posiada inną współrzędną z, to następuje przeniesienie punktów do tablicy w celu wykonania otoczki wypukłej na tych punktach i obliczeniu pola. Następnie porównywana jest dotychczasowa maksymalna wartość pola i jeśli nowy wynik jest większy od istniejącego maksimum to jest zamieniane.

@param arr - lista wszystkich pobranych punktów z bazy danych

@param i - zmienna pomocnicza do iteracji po elementach

@return - maksymalna wartość pola w obszarach

```
public double prepare(List<Float> arr, int i);
```

Rysunek 3 Deklaracja metody prepare

Deklaracja metody ***countFields*** pozwalającej obliczyć pole figury zrzutowanej na przestrzeń dwuwymiarową za pomocą wzoru do analitycznego obliczania pól.

@param i - zmienna pomocnicza wykorzystywana do iteracji

@param field - zmienna pomocnicza przechowująca wartość pola

@param stcSize - rozmiar stosu

@return - ostateczna wartość pola

```
public double countFields(int i, double field, int stcSize);
```

Rysunek 4 Deklaracja metody countFields

Deklaracja metody ***actions*** wykonującej kolejne kroki pozwalające wyznaczyć pole wielokąta przy wykorzystaniu otoczki wypukłej. Po załadowaniu elementów do tablic następuje sortowanie ich w tablicy. Potem wywoływane są metody odpowiedzialne za algorytm Grahama i następuje wypisanie wartości pola.

@param i - zmienna pomocnicza wykorzystywana do iteracji

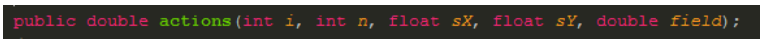
@param n - ilość wierzchołków wielokąta

@param sX - współrzędna x punktu początkowego

@param sY - współrzędna y punktu początkowego

@param field - zmienna przechowująca wartość pola

@return - pole wielokąta



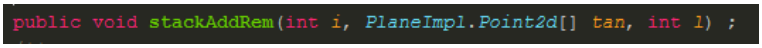
Rysunek 5 Deklaracja metody actions

Deklaracja metody ***stackAddRem*** algorytmu Grahama pobierająca każdorazowo wartość iteracyjną z tablicy (getX) i dodająca do stosu. W zależności od wyniku wyznacznika wartości są usuwane ze stosu (usuwanie punktów, które spowodowały by zbudowanie otoczki wklęsłej)

@param i - zmienna pomocnicza wykorzystywana do iteracji

@param tan - tangens nachylenia punktu do osi X

@param l - rozmiar tablicy tangens

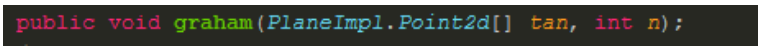


Rysunek 6 Deklaracja metody stackAddRem

Deklaracja metody ***graham*** implementującej w głównej części algorytm Grahama kierunki w których przechodzimy do kolejnego punktu tak, by zbudować otoczkę wypukłą. Do pomocy wykorzystywany jest stos.

@param tan - tangens nachylenia punktu do osi X

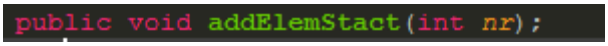
@param n - ilość wierzchołków wielokąta



Rysunek 7 Deklaracja metody graham

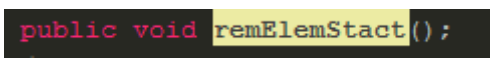
Deklaracja metody ***addElemStact*** dodającej do stosu wartość i inkrementująca ilość znajdujących się na nim elementów

@param nr - wartość do dodania na stos



Rysunek 8 Deklaracja metody addElemStack

Deklaracja metody ***remElemStact*** usuwającej wartości ze stosu i dekrementująca ilość znajdujących się na nim elementów



Rysunek 9 Deklaracja metody remElemStack

Deklaracja metody ***det*** zwracającej wartość wyznacznika ze współrzędnych punktów zawartych w tablicy współrzędnych wielokąta

@param a - indeks trzeciej współrzędnej

@param b - indeks drugiej współrzędnej

@param c - indeks pierwszej współrzędnej

@return określenie czy wyznacznik jest dodatni czy ujemny

```
public int det(int a, int b, int c);
```

Rysunek 10 Deklaracja metody det

MPlane.java

Klasa **MPlane** będąca servletem pozwalającym na połączenie między aplikacją a bazą danych

```
public class MPlane extends HttpServlet {
    private static final long serialVersionUID = 123;
    @EJB
    private IPlaneRemote plane;
    List<Float> points = new ArrayList<Float>();
```

Rysunek 11 Klasa MPlane

Metoda ***getConnection*** łącząca się z bazą danych za pomocą nazwy JNDI na serwerze Glassfish.

@param ds - nazwa JNDI (parametr poprany metodą GET)

@return - połączenie z bazą

@throws SQLException - zapewnia informacje o błędzie dostępu do bazy danych

@throws NamingException - zapewnia informacje o błędzie gdy nazwa JNDI jest niepoprawna

```
private Connection getConnection(String ds) throws
        SQLException,NamingException {
    Connection connection = null;
    InitialContext context = new InitialContext();
    DataSource dataSource = (DataSource) context.lookup(ds);
    connection = dataSource.getConnection();

    return connection;
}
```

Rysunek 12 Implementacja metody getConnection

Metoda ***addPointsToList*** pobierająca wartości punktów z bazy i zapisująca je do listy.

@param rs - zbiór rekordów zwróconych po wykonaniu zapytania w bazie danych

@throws SQLException - zapewnia informacje o błędzie dostępu do bazy danych

```
private void addPointsToList(ResultSet rs) throws SQLException{
    for (;;){
            if (rs.next()){
                points.add(rs.getFloat("x"));
                points.add(rs.getFloat("y"));
                points.add(rs.getFloat("z"));
            }else break;
        }
}
```

Rysunek 13 Implementacja metody addPointsToList

Metoda ***getPoints*** w której pobierane są wyniki zapytania w bazie danych i przekazywane są do zapisania na liście.

@param connection - referencja do otwartego połączenia z bazą danych

@param sql - treść zapytania niezbędna do pobrania rezultatu

@param response - odpowiedź servletu

@throws Exception -przechwycenie zdarzenia, które może zakłócić poprawne działanie programu

@throws IOException – w przypadku wystąpienia błędu wejścia/wyjścia

```
private void getPoints(Connection connection, String sql,
        HttpServletResponse response) throws  Exception,IOException {
    Statement stmt;
    ResultSet rs;
    if (connection != null) {
            stmt =  connection.createStatement();
            rs = stmt.executeQuery(sql);
            addPointsToList(rs);
    }
    response.setContentType("text/html;charset=UTF-8");
    response.getWriter().format(Locale.ENGLISH, "%.5f",
            plane.general(points));
}
```

Rysunek 14 Implementacja metody getPoints

Metoda ***doGet***, dzięki której po przekazaniu parametru w żądaniu URL, następuje rejestracja użytkownika i jeśli przebiegnie pomyślnie wykonywane są metody określające połączenie z bazą i pobranie danych z bazy.

@param request - żądanie servletu

@param response - odpowiedź servletu

@throws ServletException – w przypadku wystąpienia błędu związanego z servletem

@throws IOException – w przypadku wystąpienia błędu wejścia/wyjścia

```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)throws ServletException, IOException {
   //  response.getWriter().format(Locale.ENGLISH, "%.5f", 0.0);
    try {
        String connect = "java:global/ejb-project/GameManager!"
                +"pl.jrj.game.IGameRemote";

        InitialContext con = new InitialContext();
        pl.jrj.game.IGameRemote game = (pl.jrj.game.IGameRemote)
                con.lookup(connect);

        if (game.register(6, "104896")){
            getPoints(getConnection(request.getParameter("ds"))
                    ,"SELECT * FROM Otable order by z, y, x; ",response);
        }
     //   processRequest(request, response);
    } catch (NamingException ex) {
        response.getWriter().format(Locale.ENGLISH, "%.5f", 0.0);
    } catch (Exception e) {
        response.getWriter().format(Locale.ENGLISH, "%.5f", 0.0);
    }
}
```

Rysunek 15 Implementacja metody doGet

PlaneImpl.java

Klasa **PlaneImpl** pozwalająca na poszukiwanie maksymalnego pola w zależności od położenia współrzędnej Z, z dokładnością do 5 miejsc dziesiętnych.

```
@Stateless
public class PlaneImpl implements IPlaneRemote {
    List<Integer> stack = new ArrayList<Integer>();
    List<Point3d> points3d = new ArrayList<Point3d>();
    Point2d[] points;
    int pocz=0;
    float z = 0;
```

Rysunek 16 Implementacja klasy PlaneImpl

Główna metoda ***general*** wywoływana przez bean z serwletu określająca wykonywanie kolejnych metod w klasie.

@param arr - lista wszystkich pobranych punktów z bazy danych

@return - maksymalne pole przestrzeni

```
@Override
public double general(List<Float> arr) {
    double wynik;
    wynik = prepare(arr,0);
    return wynik;
}
```

Rysunek 17 Implementacja metody general

Metoda ***prepare*** pozwala na przyporządkowanie określonych wartości pobranych z bazy do listy jako współrzędne punktów w przestrzeni 3D. Jeśli kolejna grupa punktów posiada inną współrzędną z, to następuje przeniesienie punktów do tablicy w celu wykonania otoczki wypukłej na tych punktach i obliczeniu pola. Następnie porównywana jest dotychczasowa maksymalna wartość pola i jeśli nowy wynik jest większy od istniejącego maksimum to jest zamieniane.

@param arr - lista wszystkich pobranych punktów z bazy danych

@param i - zmienna pomocnicza do iteracji po elementach

@return - maksymalna wartość pola w obszarach

```
@Override
public double prepare(List<Float> arr, int i) {
    double max = 0;
    int j;
    double wynik=0;
    float flag = arr.get(2);
    int arrSize = arr.size();
    for (i=0;i<arrSize;i+=3){

        float x = arr.get(i);
        float y = arr.get(i+1);
        float zz = arr.get(i+2);

        if ((flag==zz)){
            points3d.add(new Point3d(x,y,z));
        }
        if ((flag!=zz)){
```

Rysunek 18 Implementacja metody prepare

Metoda ***countFields*** pozwalająca obliczyć pole figury zrzutowanej na przestrzeń dwuwymiarową za pomocą wzoru do analitycznego obliczania pól.

@param i - zmienna pomocnicza wykorzystywana do iteracji

@param field - zmienna pomocnicza przechowująca wartość pola

@param stcSize - rozmiar stosu

@return - ostateczna wartość pola

```
@Override
public double countFields(int i, double field, int stcSize) {
    for (i = 0;i<stcSize;i++ ){
        if (i==0){
            field+= (points[stack.get(i)].getX())*
                    (points[stack.get(i+1)].getY()-
                    points[stack.get(stack.size()-1)].getY());
        }else if (i==stack.size()-1){
            field+= (points[stack.get(i)].getX())*
                    (points[stack.get(0)].getY()-
                    points[stack.get(i-1)].getY());
        }else{
            field+= (points[stack.get(i)].getX())*
                    (points[stack.get(i+1)].getY()-
                    points[stack.get(i-1)].getY());
        }
    }
    return field;
}
```

Rysunek 19 Implementacja metody countFields

Metoda ***actions*** wykonująca kolejne kroki pozwalające wyznaczyć pole wielokąta przy wykorzystaniu otoczki wypukłej. Po załadowaniu elementów do tablic następuje sortowanie ich w tablicy. Potem wywoływane są metody odpowiedzialne za algorytm Grahama i następuje wypisanie wartości pola.

@param i - zmienna pomocnicza wykorzystywana do iteracji

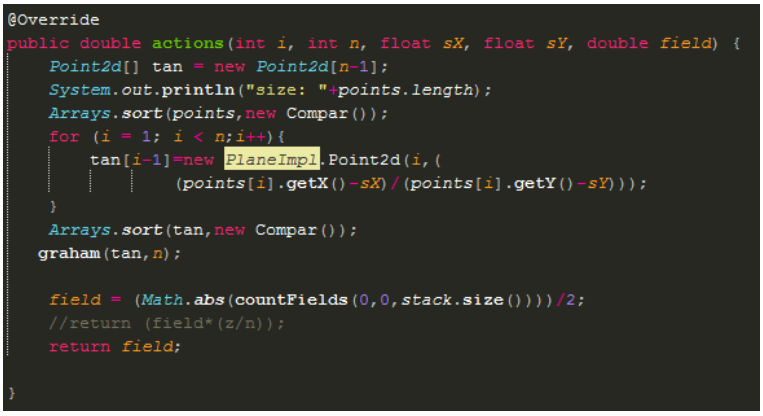
@param n - ilość wierzchołków wielokąta

@param sX - współrzędna x punktu początkowego

@param sY - współrzędna y punktu początkowego

@param field - zmienna przechowująca wartość pola

@return - pole wielokąta



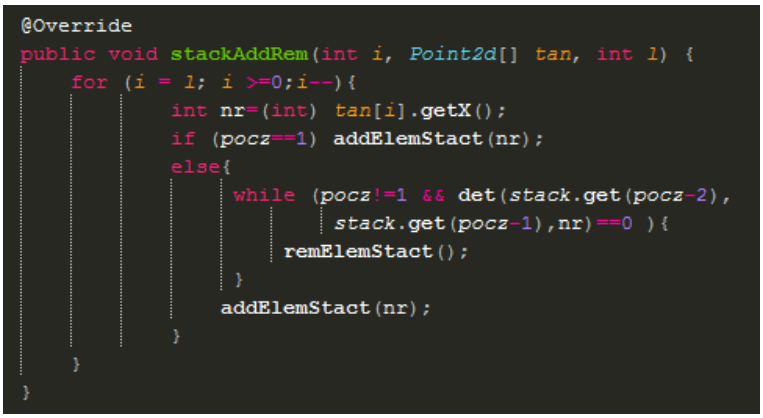
Rysunek 20 Implementacja metody actions

Metoda ***stackAddRem*** algorytmu Grahama pobierająca każdorazowo wartość iteracyjną z tablicy (getX) i dodająca do stosu. W zależności od wyniku wyznacznika wartości są usuwane ze stosu (usuwanie punktów, które spowodowały by zbudowanie otoczki wklęsłej)

@param i - zmienna pomocnicza wykorzystywana do iteracji

@param tan - tangens nachylenia punktu do osi X

@param l - rozmiar tablicy tangens



Rysunek 21 Implementacja metody stackAddRem

Metoda ***graham*** implementująca w głównej części algorytm Grahama określający kierunki w których przechodzimy do kolejnego punktu tak, by zbudować otoczkę wypukłą. Do pomocy wykorzystywany jest stos.

@param tan - tangens nachylenia punktu do osi X

@param n - ilość wierzchołków wielokąta

```java
@Override
public void graham(Point2d[] tan, int n) {
    int dett;
   addElemStact(0);
   stackAddRem(0,tan,tan.length-1);
   while (pocz!=1 && det(stack.get(pocz-2),stack.get(pocz-1),0)==0)
      remElemStact();
    if (pocz!=n)
      addElemStact(0);
}
```

Rysunek 22 Implementacja metody graham

Metoda ***addElemStact*** dodająca do stosu wartość i inkrementująca ilość znajdujących się na nim elementów

@param nr - wartość do dodania na stos

```java
@Override
public void addElemStact(int nr) {
    stack.add(nr);
    pocz++;
}
```

Rysunek 23 Implementacja metody addElemStack

Metoda ***remElemStact*** usuwająca wartości ze stosu i dekrementująca ilość znajdujących się na nim elementów

```java
@Override
public void remElemStact() {
    stack.remove(stack.size()-1);
    pocz--;
}
```

Rysunek 24 Implementacja metody remElemStack

Metoda ***det*** zwracająca wartość wyznacznika ze współrzędnych punktów zawartych w tablicy współrzędnych wielokąta

@param a - indeks trzeciej współrzędnej

@param b - indeks drugiej współrzędnej

@param c - indeks pierwszej współrzędnej

@return określenie czy wyznacznik jest dodatni czy ujemny

```java
@Override
public int det(int a, int b, int c) {
    double p=points[a].getX()*points[b].getY()+
             points[b].getX()*points[c].getY()+
             points[c].getX()*points[a].getY()-
             points[c].getX()*points[b].getY()-
             points[a].getX()*points[c].getY()-
             points[b].getX()*points[a].getY();
    if (p>=0)
        return 1;
    else
        return 0;
}
```

Rysunek 25 Implementacja metody det

Klasa wewnętrzna ***Compar*** implementująca interfejs i udostępniająca compare, która przyjmuje 2 obiekty i zawiera utworzony schemat w przypadku różnych lub równych obiektów.

```java
private class Compar implements Comparator<Point2d>{

    @Override
    public int compare(Point2d o1, Point2d o2) {
        Float y1 = o1.getY();
        Float y2 = o2.getY();
        int sComp = y1.compareTo(y2);

        if (sComp!=0){
            return sComp;
        } else{
            Float x1 = o1.getX();
            Float x2 = o2.getX();
            return x1.compareTo(x2);
        }
    }

}
```

Rysunek 26 Implementacja klasy Compar

Klasa ***Point2d*** reprezentująca zbiór punktów x,y w przestrzeni dwuwymiarowej

```java
public class Point2d {
    float x;
    float y;

    public Point2d(){}

    public Point2d (float x, float y){
        this.x = x;
        this.y = y;
    }
```

Rysunek 27 Implementacja klasy Point2d

Klasa ***Point3d*** reprezentująca zbiór punktów x,y,z w przestrzeni trójwymiarowej

```
public class Point3d {
    float x;
    float y;
    float z;

    public Point3d(){}

    public Point3d (float x, float y, float z){
        this.x = x;
        this.y = y;
        this.z = z;
    }
```

Rysunek 28 Implementacja klasy Point2d

Algorytm Grahama – wyznaczanie otoczki wypukłej

Zapewnia obliczenie otoczki wypukłej zestawu punktów na płaszczyźnie. Działa w czasie O(n log n) i używa O(n) dodatkowej pamięci.

Na początku pobieramy punkty z bazy posortowane kolejno względem Z, Y i X. Przenosimy dane do listy do momentu pojawienia się innej wartości Z. Jeśli taka się pojawi to zbiór współrzędnych przenoszony jest do tablicy pomocniczej, dzięki której liczona jest otoczka wypukła. Po jej wyznaczeniu liczone jest pole. Następnie program określa czy pole jest większe od aktualnie maksymalnego. Jeśli tak, to zamienia na nową wartość maksymalnego pola. Po wyznaczeniu wszystkich obszarów, do servletu zwracana jest wartość maksymalnego pola, która następnie wypisywana jest na ekran jako wynik końcowy.