Actividad 4 Una empresa de criptomonedas tiene un agente financiero que visita diferentes nodos de la red (equivalente a sucursales) para realizar cobros y pagos en criptomonedas. Cada vez que inicia un recorrido, cuenta con un saldo inicial en su billetera digital. Cada nodo tiene un monto fijo que el agente puede cobrar (+) o pagar (-) en criptomonedas. El agente debe planificar su recorrido de manera que nunca se quede sin fondos al llegar a un nodo que requiera un pago. Para decidir qué nodo visitar, consulta una estadística de transacciones y prioriza aquellos nodos donde pueda cobrar más criptomonedas. Si no tiene saldo suficiente para continuar, busca la siguiente mejor opción. Además, el agente debe considerar la volatilidad del mercado: el valor de las criptomonedas puede cambiar durante el recorrido, lo que afecta su saldo en términos de dólares (u otra moneda fiat). Por lo tanto, el agente debe intentar maximizar su saldo en dólares al final del recorrido.

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

import java.util.Random;

// Principio de Responsabilidad Única (SRP)

class Nodo {

private String nombre;

private double montoCripto; // Monto en criptomonedas

public Nodo(String nombre, double montoCripto) {

this.nombre = nombre;

this.montoCripto = montoCripto;

}

public String getNombre() {

return nombre;

}

public double getMontoCripto() {

return montoCripto;

}

}

// Principio de Abierto/Cerrado (OCP)

class PlanificadorRecorrido {

private double saldoInicialCripto;

private List<Nodo> nodos;

private double tasaCambioActual; // Tasa de cambio cripto a dólares

public PlanificadorRecorrido(double saldoInicialCripto, List<Nodo> nodos, double tasaCambioActual) {

this.saldoInicialCripto = saldoInicialCripto;

this.nodos = new ArrayList<>(nodos);

this.tasaCambioActual = tasaCambioActual;

}

// Heurística: Ordenar nodos por monto de cobro (de mayor a menor)

private void ordenarNodos() {

Collections.sort(nodos, Comparator.comparingDouble(Nodo::getMontoCripto).reversed());

}

// Simular la volatilidad del mercado

private double simularVolatilidad(double tasaCambio) {

Random rand = new Random();

// Simular un cambio aleatorio entre -10% y +10%

return tasaCambio \* (1 + (rand.nextDouble() - 0.5) \* 0.2);

}

// Planificar recorrido

public List<Nodo> planificarRecorrido() {

ordenarNodos();

List<Nodo> recorrido = new ArrayList<>();

double saldoActualCripto = saldoInicialCripto;

double saldoActualDolares = saldoInicialCripto \* tasaCambioActual;

for (Nodo nodo : nodos) {

double nuevaTasaCambio = simularVolatilidad(tasaCambioActual);

double montoDolares = nodo.getMontoCripto() \* nuevaTasaCambio;

if (saldoActualCripto + nodo.getMontoCripto() >= 0) {

recorrido.add(nodo);

saldoActualCripto += nodo.getMontoCripto();

saldoActualDolares += montoDolares;

System.out.println("Visitando: " + nodo.getNombre() +

", Saldo Cripto: " + saldoActualCripto +

", Saldo Dólares: " + saldoActualDolares);

}

}

return recorrido;

}

}

// Principio de Sustitución de Liskov (LSP) y Principio de Segregación de Interfaces (ISP)

interface Reporte {

void generarReporte(List<Nodo> recorrido);

}

class ReporteTexto implements Reporte {

@Override

public void generarReporte(List<Nodo> recorrido) {

System.out.println("Recorrido planificado:");

for (Nodo nodo : recorrido) {

System.out.println("Nodo: " + nodo.getNombre() + ", Monto Cripto: " + nodo.getMontoCripto());

}

}

}

// Principio de Inversión de Dependencias (DIP)

public class Main {

public static void main(String[] args) {

// Crear nodos

List<Nodo> nodos = new ArrayList<>();

nodos.add(new Nodo("Nodo A", 1.5));

nodos.add(new Nodo("Nodo B", -0.5));

nodos.add(new Nodo("Nodo C", 2.0));

nodos.add(new Nodo("Nodo D", -1.0));

nodos.add(new Nodo("Nodo E", 1.0));

// Tasa de cambio inicial: 1 cripto = 500 dólares

double tasaCambioInicial = 500;

// Planificar recorrido con saldo inicial de 2 criptomonedas

PlanificadorRecorrido planificador = new PlanificadorRecorrido(2, nodos, tasaCambioInicial);

List<Nodo> recorrido = planificador.planificarRecorrido();

// Generar reporte

Reporte reporte = new ReporteTexto();

reporte.generarReporte(recorrido);

}

}