Händelsestyrd simulering

Inledning

Du skall konstruera ett program som simulerar vad som händer när kunder kommer till ett bankkontor med flera kassor.

Om en kund anländer och någon av kassorna är ledig betjänas han omgående. Annars får kunden vänta i en kö som är gemensam för alla kassor. När någon av kassorna blir ledig och kön inte är tom blir den kund som står först i kön betjänad.

För att kunna göra beräkningar eller simuleringar av hur systemet fungerar behövs modeller för när kunder anländer och hur lång tid en betjäning tar. Vi antar att alla kassor är ekvivalenta och att inga kunder lämnar lokalen utan att ha blivit betjänade.

Vi antar att ankomst- och betjäningstiderna är slumpmässiga och oberoende av varandra. För att kunna göra en simulering behöver vi veta vilka statistiska fördelningar dessa tider har. Vi förutsätter att det finns objekt av typen `RandomDouble` som har en metod `double next()` som ger slumpmässiga värden på tiden mellan två ankomster och slumpmässiga betjäningstider.

Man skulle kunna genomföra simuleringen genom att först skapa alla kunder med deras ankomst- och betjäningstider och sedan stega fram tiden med 1 sekund i taget och undersöka om det händer något och i så fall rapportera detta och ändra de attribut som beskriver tillståndet för bankkontoret.

Det är mer rationellt att låta simuleringen styras av händelserna så att vi i varje simuleringssteg ställer fram klockan till tiden för nästa händelse. De händelser som gör att tillståndet förändras är att en kund anländer till banken och att en kund lämnar den efter att ha blivit betjänad.

Tillståndet för banken beskrivs av en tidpunkt, antalet lediga kassor, en kö av kunder som väntar på betjäning och en mängd av händelser som skall inträffa.

Simuleringen startar med att man skapar den första händelsen som är ankomsten för den första kunden och lägger den i händelsemängden. I fortsättningen tar man bort den händelse i händelsemängden som har den tidigaste tidpunkten och låter den påverka tillståndet i banken.

Om händelsen är en ankomst och det finns någon ledig kassa får kunden betjäning direkt och händelsen att denna kund lämnar baken läggs till händelsemängden. Med hjälp av slumptidsgeneratorn för betjäningstider kan man räkna ut när kunden kommer att färdig och lämna banken. Denna framtida händelse lägges till händelsemängden. Antalet lediga kassor minskar med 1.

Om det inte finns någon ledig kassa placeras kunden i kundkön. I båda fallen skapas en ny ankomsthändelse som lägges till händelsemängden.

När en kund lämnar banken finns också två alternativ. Om kön är tom ökas antalet lediga kassor med 1. Annars påbörjas betjäning av den kund som står först i kön och den händelse som innebär att denna kund lämnar systemet tillfogas händelsemängden.

Vi varje händelse skall programmet göra en utskrift som visar tiden, kundens nummer, vilken sorts händelse det är och kundkön efter händelsen.

Simuleringen avslutas efter en timme och den maximala kölängden skrivs ut.

Exempel

Följande utskrift visar början av en simulering med 3 kassor

```
tid     kund    händelse  kö
3       0       ankomst   []
31      1       ankomst   []
31      2       ankomst   []
34      3       ankomst   [3]
35      2       avgång    []
56      3       avgång    []
83      4       ankomst   []
94      5       ankomst   [5]
163     6       ankomst   [5, 6]
176     4       avgång    [6]
196     7       ankomst   [6, 7]
204     8       ankomst   [6, 7, 8]
209     9       ankomst   [6, 7, 8, 9]
215     1       avgång    [7, 8, 9]
271     0       avgång    [8, 9]
```

När simuleringen börjar finns bara ankomsten för kund nr 0 i händelsemängden.

```
3       0       ankomst
```

När en ankomsthändelse påverkar systemet skapas nästa ankomsthändelse och avgångshändelsen för kund 0 så att händelsemängden innehåller

```
31      1       ankomst
271     0       avgång
```

Ankomsten av kund nr 1 gör att händelsemängden får innehållet

```
31      2       ankomst
215     1       avgång
271     0       avgång
```

När kund nr 2 kommit innehåller den

```
34      3       ankomst
35      2       avgång
215     1       avgång
271     0       avgång
```

När kund 3 kommer är alla kassor upptagna och kunden får vänta i kön. Händelsemängden innehåller

```
35      2       avgång
83      4       ankomst
215     1       avgång
271     0       avgång
```

När kund nr 2 blir färdig får kund nr 3 komma till kassan:

```
56      3       avgång
83      4       ankomst
215     1       avgång
271     0       avgång
```

Vi noterar att det alltid finns exakt en ankomsthändelse i händelsemängden och att det kan finnas högst så många avgångshändelser som det finns kassor.

Fundera på vilka klasser som bör finnas och vilka attribut de skall ha innan du läser nästa avsnitt! Vilka klasser i `java.util` kan användas?

Programdesign – klasser

Det bör finnas en klass, `Bank`, som representerar tillståndet i banken. Den skall ha tiden, antalet lediga kassor, kundkön och händelsemängden som attribut. Det kan också vara lämpligt att låta de två slumptidsgeneratorerna och sluttiden för simuleringen vara attribut i klassen.

För att representera kundkön kan vi använda `ArrayDeque` från `java.util`. Det kan vara praktiskt att låta kön hålla reda på sin maximala längd. Därför utvidgar vi `ArrayDeque` och tillfogar ett attribut för ändamålet.

Den klass som skall representera en kund behöver bara ett attribut, kundens nummer.

Det är lämpligt att representera mängden av framtida händelser med en prioritetskö där prioriteten ges av tidpunkten när händelsen skall inträffa. Vi behöver nämligen komma åt framtida händelser i ordning efter den tidpunkt när de skall inträffa. Det finns en prioritetskö i `java.util`.

Elementen i prioritetskön skall naturligtvis vara händelser. Eftersom det finns två sorters händelser, ankomster och avgångar, som skall påverka tillståndet i banken på olika sätt är det lämpligt att ha en klass för vardera sorten med en gemensam superklass. En händelse karaktäriseras av tidpunkten och den kund som är involverad. Attributen kan placeras i superklassen. Eftersom elementen skall kunna ordnas efter prioritet skall superklassen implementera gränssnittet `Comparable`. Det behövs en metod som returnerar tiden.

Slumptidsgeneratorerna skall implementera gränssnittet

```
public interface RandomDouble {
    public double next();
}
```

Om kunderna ankommer slumpmässigt och oberoende av varandra med en fix genomsnittlig frekvens, λ kunder/tim, så kommer tiden mellan ankomsterna att vara vad som i statistiken kallas för *exponentialfördelad*.

Studier av betjäningstider i verkliga system visar att de ofta också är exponentialfördelade med en parameter, μ, som anger hur många kunder/tim som i genomsnitt kan hanteras av en betjäningsstation. Om vi har k stycken kassor måste λ vara mindre än $k\mu$ för att kön inte skall växa över alla gränser.

Det är enkelt att skriva en metod som ger slumptal som är exponentialfördelade som kan användas för att simulera vad som händer i banken.

```
import java.util.Random;

public class ExponentialRandom implements RandomDouble {
    private double lambda;
    private Random random;

    public ExponentialRandom(double lambda) {
        this.lambda = lambda;
        random = new Random();
    }

    public ExponentialRandom(double lambda, long seed) {
        this.lambda = lambda;
        random = new Random(seed);
    }

    public double next() {
        return -Math.log(1 - random.nextDouble()) / lambda;
    }
}
```

Metoden `next` använder slumptalsgeneratorn i `java.util.Random` som ger slumptal i intervallet $[0, 1)$. När man skapar denna kan man antingen ange ett startvärde, `seed`, som gör att man får samma sekvens av slumptal vid varje exekvering eller underlåta detta varvid tiden för skapandet

bestämmer startvärdet. Den som läst statistik kan känna igen inversen till fördelningsfunktionen för exponentialfördelningen.

Innan du läser nästa avsnitt skall du fundera över vilka metoder och konstruerare som klasserna bör tillhandahålla.

Programdesign – konstruerare och metoder

Bankklassen bör har en konstruererar med vars hjälp man kan ge antalet kassor, slumptidsgeneratorerna för tider mellan ankomster och betjäningstider och eventuellt också sluttiden för simuleringen.

Det behövs en metod som utför simuleringen genom att hämta händelser ur händelsemängden, ser till att händelserna påverkar tillståndet och rapporterar vad som hänt.

När en händelse skall hanteras behövs metoder som utför de tillståndsförändringar som skall genomföras. Det behövs en metod för händelsen att en kund anländer till och en för att en kund lämnar bankkontoret. Ankomstmetoden behöver kunden som parameter.

Det är lämpligt att placera `main`-metoden i bankklassen. Den skall skapa slumptidsgeneratorerna, banken och anropa den metod som utför simuleringen.

Kundkön tillhandahåller metoder för att lägga till och ta bort ett element ur kön, ta reda på antalet element i kön och avgöra om den är tom. Sådana metoder finns i gränssnittet `java.util.Queue` och är implementerade i `ArrayDeque`. Eftersom kön skall hålla reda på sin maximala längd behöver man skugga metoden `offer` och lägga till en metod som returnerar den maximala längden. Det behövs ingen särskild metod för att skriva ut innehållet i kön; metoden `toString` ger en sträng med alla elementen.

Prioritetskön i `java.util` går att använda som den är.

Kundklassen innehåller bara ett heltal. Vi behöver skriva ut det när vi rapporterar händelser. En `toString()`-metod som returnerar talet som en sträng är allt vi behöver.

Den abstrakta händelseklassen skall ha en konstruerare med tidpunkten och kunden som parametrar. Den skall implementera metoden `compareTo`, ha en metod som returnerar tiden och gärna en `toString()` som returnerar en sträng med tidpunkten och kunden. Den metod med vilken en händelse skall påverka bankens tillstånd skall implementeras olika i de två subklasserna, men måste deklareras här. Den behöver banken som parameter.

Subklasserna skall implementera `toString()` så att resultatet kan användas i utskrifter av händelsen. Metoden kan använda `toString()` från superklassen för att bygga upp hela händelsesträngen.

Implementering

Kunden får sitt nummer i konstrueraren i kundklassen med hjälp av med en statisk `int`-variabel som håller reda på antalet skapade kunder

Kundkön och prioritetskön skall naturligtvis bara kunna innehålla kunder resp. händelser.

Om man vill reproducera resultaten i exemplet skall man använda följande slumptalsgeneratorer

```
RandomDouble arrival = new ExponentialRandom(100, 4321);
RandomDouble service = new ExponentialRandom(40, 8765);
```

där tidsenheten är timmar. Utskrifterna anger tiderna i sekunder.

Köteori

Det finns en omfattande matematisk teori för köer. Det kösystem vi simulerar i denna uppgift kallas i litteraturen för ett M/M/k väntsystem, där M står för Markov och representerar den stokastiska modell vi använt för ankomst- och betjäningstider medan k står för antalet kassor. Systemet är så enkelt att man inte behöver simulera utan det går ganska enkelt att beräkna sannolikheten för att en kund behöver vänta och väntevärden för kölängd och kötid.

Simuleringen kommer förmodligen att visa att det går mycket snabbare att skapa en lång kö än att avveckla den.

1 Implementering

```
package simulation;
import java.util.PriorityQueue;

public class Bank {
    private int idle;
    private double time;
    private double finalTime;
    private RandomDouble arrival;
    private RandomDouble service;
    private CustomerQueue queue = new CustomerQueue();
    private PriorityQueue<Event> eventQueue = new PriorityQueue<Event>();

    public Bank(int counters, double finalTime, RandomDouble arrival,
            RandomDouble service) {
        idle = counters;
        this.arrival = arrival;
        this.service = service;
        this.finalTime = finalTime;
    }

    void arrival(Customer customer) {
        eventQueue.add(new Arrival(time + arrival.next()));
        if (idle > 0) {
            idle--;
            eventQueue.add(new Departure(time + service.next(), customer));
        } else {
            queue.offer(customer);
        }
    }

    void departure() {
        idle++;
        if (!queue.isEmpty()) {
            Customer customer = queue.poll();
            idle--;
            eventQueue.add(new Departure(time + service.next(), customer));
        }
    }
```

```
    public void simulate() {
        System.out.println("tid\tkund\thändelse  kö");
        eventQueue.add(new Arrival(arrival.next()));
        while (time < finalTime) {
            Event event = eventQueue.poll();
            time = event.time();
            event.execute(this);
            System.out.print(event);
            System.out.println("\t  " + queue);
        }
        System.out.println("maximal kölängd = " + queue.maximalSize());
    }

    public static void main(String[] argv) {
        RandomDouble arrival = new ExponentialRandom(100, 4321);
        RandomDouble service = new ExponentialRandom(40, 8765);
        Bank bank = new Bank(3, 1, arrival, service);
        bank.simulate();
    }
}
```

```
package simulation;
class Customer {
    private static int total = 0;
    private int id;

    Customer() {
        id = total;
        total++;
    }

    public String toString() {
        return String.valueOf(id);
    }
}
```

```
package simulation;
import java.util.ArrayDeque;

class CustomerQueue extends ArrayDeque<Customer> {
    private int max = 0;

    int maximalSize() {
        return max;
    }

    public boolean offer(Customer customer) {
        boolean result = super.offer(customer);
        max = Math.max(max, size());
        return result;
    }
}
```

När man ser innehållet i kundkön inser man att om kunderna bara innehåller information om sitt nummer så kan man representera kön med två heltal, antalet köande och numret för den första kunden i kön.

```
package simulation;
abstract class Event implements Comparable<Event> {
    private double time;
    protected Customer customer;

    Event(double time, Customer customer) {
        this.time = time;
        this.customer = customer;
    }

    public int compareTo(Event other) {
        return Double.compare(time, other.time);
    }

    double time() {
        return time;
    }
    public String toString() {
        return ((int) (time * 3600))+ "\t"  + customer;
    }

    abstract void execute(Bank bank);
}
```

```
package simulation;
class Arrival extends Event {
    Arrival(double arrivalTime) {
        super(arrivalTime, new Customer());
    }

    void execute(Bank bank) {
        bank.arrival(customer);
    }

    public String toString() {
        return super.toString() + "\tankomst";
    }
}
```

```
package simulation;
class Departure extends Event {
    Departure(double departureTime, Customer customer) {
        super(departureTime, customer);
    }

    void execute(Bank bank) {
        bank.departure();
    }

    public String toString() {
        return super.toString() + "\tavgång";
    }
}
```

```
package simulation;
public interface RandomDouble {
    public double next();
}
```

```
package simulation;
import java.util.Random;

public class ExponentialRandom implements RandomDouble {
    private double lambda;
    private Random random;

    public ExponentialRandom(double lambda) {
        this.lambda = lambda;
        random = new Random();
    }

    public ExponentialRandom(double lambda, long seed) {
        this.lambda = lambda;
        random = new Random(seed);
    }

    public double next() {
        return -Math.log(1 - random.nextDouble()) / lambda;
    }
}
```