Simulering under press

En jämförelse av tidsstyrd och händelsestyrd simulering

M A R K U S P E T T E R S S O N

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2012

Simulering under press

En jämförelse av tidsstyrd och händelsestyrd simulering

M A R K U S P E T T E R S S O N

DD143X, Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik 300 högskolepoäng
Kungliga Tekniska Högskolan år 2012
Handledare på CSC var Henrik Eriksson
Examinator var Mårten Björkman

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2012/pettersson_markus_K12055.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Sammanfattning

Denna rapport är en jämförelse mellan paradigmerna händelsestyrd och tidsstyrd simulering. En modell av en hiss i en kontorsbyggnad definieras samt implementeras i de olika paradigmerna, med hjälp av tidtagning och resultat dras sedan slutsatser om de olika paradigmernas styrkor och svagheter att simulera det aktuella systemet. Det kan konstateras att hissens tydligt diskreta egenskaper gör att systemet lämpar sig mycket bättre att implementeras händelsestyrt sett ur både enkelhet, prestanda och intuitivitet. Däremot framkommer inga tydliga skillnader i tidsmätningen av implementationen.

Abstract

This report is a comparison between the next-event and time-slicing simulation techniques. A model of an elevator in an office building is defined and implemented using the two paradigms; conclusions concerning strengths and weaknesses of the paradigms are thereafter drawn based on implementation time-taking and simulation results. It's concluded that the obvious discrete properties of the elevator makes it much easier, faster and more intuitive to be implemented with next-event technique. However, no obvious diversities in the implementation time-taking are noticed.

Innehållsförteckning

1. Introduktion

1.1. Bakgrund

Grundidén inom simuleringsteknik är att utifrån ett *system* skapa en *modell.* System kan delas in i två huvudkategorier: diskreta system och kontinuerliga system, där ett diskret system har egenskaper som ändras stegvis, t.ex. en kö där antalet personer i kön måste vara ett heltal. Ett kontinuerligt system är motsatsen: ett system där egenskaper ändras utan specifika steg, såsom vattennivån i ett tidvattenpåverkat vattendrag. Många system har dock både kontinuerliga och diskreta egenskaper – i tidvattenexemplet kan en egenskap vara om vattnet stiger eller sjunker vilket kan ses som en diskret egenskap – men de allra flesta system har övervikt på endera egenskapstypen.

> *"Few systems in practice are wholly discrete or continuous, but since one type of change predominates for most systems, it will usually be possible to classify a system as being either discrete or continuous"* (Law and Kelton, 2000, citerat i Banks et al., 2005, s. 11)

Det system som studeras i denna rapport är ett system av hissar i en kontorsbyggnad, vilket uppenbarligen till största delen har diskreta egenskaper. Kontinuerliga system kommer därför bara att studeras marginellt.

När en simulering av ett diskret system ska göras är den avgörande faktorn hur tiden ska hanteras. Händelsestyrd simulering hanterar tiden kontinuerligt så att en händelse kan ske vid en specifik tidpunkt, beroende på systemets tillstånd i den tidpunkten kommer händelsen att påverka systemet på ett visst sätt. Den engelska termen för detta är *next-event technique* (Carvalho & Luna, 2002).

Vid tidsstyrd simulering delas istället tiden upp i diskreta, jämnstora intervall. Vid varje intervall kan händelser antingen påbörjas, fortgå eller avslutas. Denna metod kan också användas för simulering av kontinuerliga system (Ziegler et al., 2000, s. 50-55), längden på intervallen kan anpassas för att uppnå önskad noggrannhet. På engelska kallas denna metod för *time-slicing* (Carvalho & Luna, 2002).

För kontinuerliga system nämns även en tredje paradigm vilken baseras på differentialekvationer som ställs upp och löses (Ziegler et al., 2000). Denna paradigm skiljer sig avsevärt från de två tidigare, då den är analytisk istället för de två tidigare som är numeriska (Banks et al., 2005).

1.2. Fördelningar

En viktig del i simulering av betjäningssystem – vilket en hiss kan betraktas som – är olika former av ankomstintensitet. En vanlig fördelning för ankomstintensitet är Poissonfördelningen. Denna har den viktiga egenskapen "minneslöshet", dvs. att sannolikheten för en ankomst är lika stor oavsett tidpunkt för den senaste ankomsten (Blom, Enger, Englund, Grandell & Holst, 2005).

En annan viktig fördelning är exponentialfördelningen, vilken beskriver längden mellan händelser i en

Poissonprocess. Exponentialfördelade slumptal kan genereras med inversmetoden (Blom et al., 2005). En exponentialfunktion y kan definieras som:

```
y = -log(rand(1))/λ
```

där `rand(1)` genererar ett slumptal mellan 0 och 1 och `λ` är ankomstintensiteten (Blom et al., 2005).

En tredje fördelning är normalfördelningen, en kontinuerlig fördelning som är vanlig för att beskriva variationen av olika företeelser. Den skulle t.ex. kunna beskriva längden hos KTH-studenter eller mätfel som summan av ett antal oberoende felkällor (Blom et al., 2005). Att generera normalfördelade slumptal är ganska bökigt. Programspråket Javas inbyggda funktion för detta, *nextGaussian()*, använder sig av Box-Müllers metod som utnyttjar polära koordinater (Oracle, 2011). Då implementationen kommer att använda den färdiga funktionen beskrivs inte metoden ytterligare här.

Både funktionen *nextGaussian()* och den inbyggdaJava-funktionen *nextDouble()* som motsvarar `rand(1)` ovan genererar inte ”riktiga” slumptal utan bara pseudoslumptal (Oracle, 2011). Detta är viktigt att notera då det är en möjlig felkälla för simuleringsresultaten.

1.3. Syfte och problemformulering

Syftet med rapporten är att jämföra de två simuleringsparadigmerna tidsstyrd och händelsestyrd simulering genom att implementera en modell för vardera paradigm. Det system som modelleras är en hiss i en kontorsbyggnad. Jämförelser kommer sedan att göras baserat på tidtagning av de båda implementationerna, personliga iakttagelser samt bedömning av möjligheten att bygga ut modellen vidare. För att verifiera att modellerna är lika kommer även en del testkörningar att göras, men inga slutsatser om hissars beteende kommer att dras av utdata från någon av modellerna.

2. Metod

I metoddelen presenteras hur simuleringen av en hiss i en kontorsbyggnad har gjorts. Arbetssättet är inspirerat av den metod som de facto är industristandard (Banks et al., 2005; Sandblad, n.d.), men har anpassats till projektets begränsade omfattning. En ytterligare faktor som påverkar arbetssättet är att denna simulering görs för att studera själva simuleringen, något egentligt mål med simuleringen finns inte.

2.1. Problemspecifikation

Då modellen inte på något sätt behöver vara en korrekt modellering av systemet har ett antal approximationer gjorts. Här beskrivs dessa samt specifikationer för modellens in- och utdata.

2.1.1. Approximationer

En verklig kontorsbyggnad har många fler parametrar än vad som är rimligt att ha med i ett litet projekt i studiesyfte. De viktigaste approximationerna som gjorts listas nedan, dessa bör absolut tas i beaktning vid en mer rigorös simulering.

- Hissens inbromsning och acceleration från/till marschfart går på nolltid
- Hissen används endast vid dagens början för färd uppåt och vid dagens slut för färd nedåt, inga resor sker mellan andra våningar och/eller vid andra tider
- Antalet personer som arbetar på respektive våning är likafördelat
- Varje på- och avstigning tar lika lång tid för varje person

2.1.2. Indata

Modellen ska kunna påverkas genom följande parametrar:

- Antalet våningar i byggnaden
- Totalt antal personer som arbetar i byggnaden
- Hissens maximala kapacitet räknat i antal personer
- Tid för en passagerare att kliva in/ur en hiss
- Hissarnas hastighet uppåt och nedåt räknat i antal sekunder per våning
- Genomsnittlig arbetstid samt standardavvikelsen för denna
- Förväntad ankomstintensitet till arbetsplatsen för olika tider under dagen

2.1.3. Utdata

Från modellen ska följande utdata kunna erhållas:

- Genomsnittlig väntetid från knapptryckning till påstigning
- Tidpunkt och längd för dagens längsta väntetid

2.2. Modellkonstruktion

Att skapa en tillräckligt bra modell är centralt för all simulering. Vad som är tillräckligt bra definieras vanligen av de specificerade kraven på resultatet. Då inga sådana formella krav ställts anses tillräckligt bra istället vara en modell som genererar en rimlig approximation av beteendet hos en verklig hiss.

2.2.1. Hissens beteende

I diagram 1 specificeras ett typiskt rörelsemönster för en hiss.

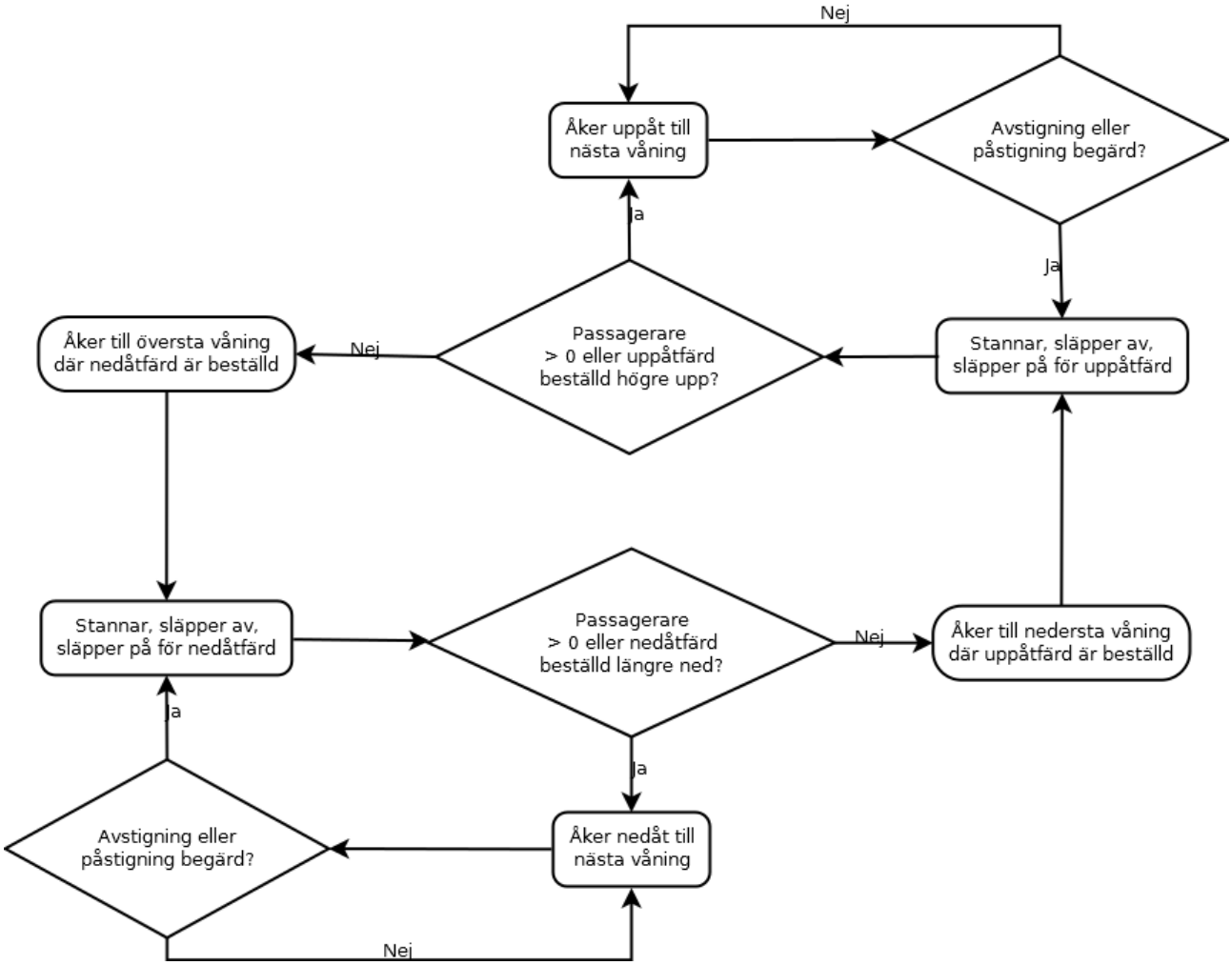

Diagram 1: UML-diagram över hissens beteende.

Underförstått i modellen är att då ingen vill åka så är hissen ledig, parkerad på det senast besökta våningsplanet. Så fort en knapp trycks in kommer hissen så fort som möjligt antingen hamna i tillståndet ”Stannar, släpper av, släpper på för nedåtfärd” eller ”Stannar, släpper av, släpper på för uppåtfärd”. När hissen så småningom töms på folk och inga fler knappar är tryckta blir hissen återigen ledig.

2.2.2. Arbetarnas beteende

Som specificerat i kapitel 2.1 har arbetarna vissa egenskaper, vilka specificeras enligt följande:

- Arbetstid – normalfördelad med specificerat väntevärde
- Starttid för arbetsdagen – Poissonfördelad mellan kl 07.00 och tre timmar framåt

2.3. Validering

En hiss är ett system som alla kommer i kontakt med då och då, därför är beteendemönstret för själva hissen lätt att validera. Vidare gör modellens tydliga avgränsning att det lätt går att verifiera att samtliga parametrar i specifikationen används i modellen. Genom detta kan konstateras att modellen är en korrekt – om än något förenklad – modell av systemet.

2.4. Programmering

Utifrån den modell som beskriver det simulerade systemet ska nu två programmeringsmodeller skapas, en för varje paradigm. Då deras beteende skiljer sig avsevärt åt kommer de att behandlas separat.

2.4.1. Händelsestyrd modell

Som beskrivits i bakgrunden handlar händelsestyrd simulering om att specificera händelser med viss starttid och varaktighet. Dessa utförs sedan i ordning så att modellen påverkas på rätt sätt.

De händelser som kan förekomma är:

- En knapp utanför hissen trycks in.
- En hiss ankommer till ett våningsplan.
- En hiss lämnar ett våningsplan.

Att en knapp trycks in kan ske av anledningarna att en person kommer till arbetsplatsen eller har arbetat färdigt för dagen.

Programmet bör alltså fungera enligt följande:

- För varje arbetare beräknas ankomsttid och sluttid för dagen.
- Hissens startposition sätts till bottenvåningen.
- När en hissknapp trycks in kommer hissen genast att åka till aktuell våning om hissen är ledig, annars jobbar hissen som specificerat i diagram 1. För varje tillstånd beräknas den tid det kommer att ta för hissen att ta sig till nästa tillstånd, tidpunkten då det nya tillståndet är uppfyllt läggs in som en händelse.

Rent implementationsmässigt görs händelsestyrningen lättast med en prioritetskö sorterad efter tid så att nästa händelse alltid ligger överst i kön. För enkelhetens skull sätts hissens startposition till bottenvåningen.

2.4.2. Tidsstyrd modell

En tidsstyrd modell ska utgå från samma hissbeteende, men kommer ha en helt annan programstruktur. För varje tidssteg ska följande göras:

- Undersök om nya personer kommit till arbetsplatsen (bottenvåningen) samt om arbetsdagen är slut för en eller flera personer på varje våning.
- Uppdatera hissens position om den är mellan två våningsplan alternativt uppdatera antalet personer som är i hissen.

Det senare kräver att det tidssteg som används är en gemensam delare till den tid det tar för hissen att förflytta sig ett våningsplan samt att lasta av/på en passagerare, detta för att försäkra att hissen stannar på varje våningsplan (se illustration 1). Detta görs enklast genom att använda tidssteget 1 och sedan sätta alla tider till heltal.

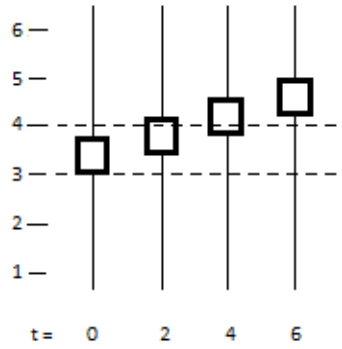

Illustration 1: Om tidssteget dt inte är en delare till tiden t_p för att passera en våning kan våningsplan hoppas över. Här är dt = 2 och t_p = 5, vilket gör att våning 4 aldrig besöks.

Värt att notera är att många tidssteg kan bli händelselösa, t.ex. då hissen är ledig.

2.5. Verifiering och testning

Det enklaste sättet att testa att implementationerna blivit rätt är att jämföra dem mot varandra. Ett optimalt resultat vore att båda ger identiska resultat, men då vissa approximationer förekommer är det inte möjligt. Dock bör resultaten efter ett stort antal körningar konvergera mot samma resultat. En möjlighet är att båda implementationerna innehåller samma fel och därmed får likadana resultat, men då simuleringsresultaten inte ska användas i verkligheten kan den risken bortses från.

2.6. Experimentplanering, utförande samt resultatanalys

I en simulering med annat syfte än att enbart studera implementationen ska givetvis den konstruerade modellen användas för att genomföra experiment vars resultat sedan tolkas. Då detta projekt uteslutande är till för att jämföra svårigheten att implementera samma modell på två olika sätt har inga experiment utförts, några resultat utöver de som använts för att verifiera de båda implementationernas korrekthet finns inte heller.

3. Resultat

Två enkla modeller implementerades i Java (för källkod se appendix). Här presenteras resultat av testkörningar, implementationstid samt personliga iakttagelser från implementationen.

3.1. Testkörningar

För att jämföra de båda implementationerna gjordes fyra olika simuleringar femtio gånger vardera för varje implementation. Medianvärdet för maximal respektive genomsnittlig väntetid beräknades för vart och ett av de fyra försöken, dessa data redovisas nedan tillsammans med indata.

Parameter	Försök 1	Försök 2	Försök 3	Försök 4
Antal våningar	2	5	10	20
På/avstigningstid (sekunder)	10	10	2	1
Hastighet (sekunder/våning)	10	15	5	2
Maxkapacitet (antal personer)	1	3	4	15
Arbetstid – väntevärde (timmar)	8	8	8	8
Arbetstid – standardavvikelse (minuter)	0	30	30	60
Antal anställda	1	50	200	1500
Medelväntetid tidsstyrd	3	43	34	62
Medelväntetid händelsestyrd	4	43	35	68
Maxväntetid tidsstyrd	10	157	239	663
Maxväntetid händelsestyrd	10	168	266	676

3.2. Implementation

3.2.1. Tidtagning

Under arbetet med implementationerna mättes den total redigeringstiden av koden. Den händelsestyrda modellen implementerades först, under totalt 9 timmar och 25 minuter. Därefter implementerades den tidsstyrda modellen under totalt 8 timmar och 32 minuter. I båda implementationerna uppkom ett par små men svårupptäckta buggar, felsökningstiden av dessa var uppskattningsvis lika stor för båda implementationerna.

3.2.2. Personliga iakttagelser

Det var till en början svårt att tänka händelsestyrt, svårigheten bestod framförallt i att kunna förutse tidpunkten för framtida händelser. När väl tröskeln var passerad var händelsestyrd simulering ganska intuitivt; så fort programmet vet att något ska hända läggs en händelse till vid rätt tidpunkt. Detta blir

också mycket effektivt då alla händelselösa tidssteg helt enkelt hoppas över.

Tidsstyrd simulering är betydligt mer omständlig att implementera, eftersom programmet vid varje tidssteg måste kontrollera precis hela systemets tillstånd för att veta vad som ska göras. Detta syns inte minst i koden som både innehåller fler variabler och fler rader kod.

Något som underlättade arbetet med båda simuleringarna var att hela tiden arbeta för att implementera modellen och ingenting annat. Den något haltande programstruktur som båda implementationerna lider av uppkom till stor del just på att avsteg från modellspecifikationen togs. Även i de loggar som fördes under arbetet syns det att många problem längs vägen hade kunnat undvikas genom att strikt följa modellen. Denna iakttagelse handlar snarare om simulering överlag än om en specifik paradigm, men den är ändå värd att notera.

4. Diskussion

Resultaten från testkörning visar att de båda implementationerna ger ungefär samma utdata, vilket indikerar att båda modellerna bör vara korrekt implementerade. Värdena från den händelsestyrda modellen är något högre, vilket troligen beror på att dess generering av ankomsttider tillåter flera ankomster varje sekund. Anledningen är att det är tiden till nästa ankomst som beräknas vilken kan vara mycket liten. Den tidsstyrda modellen beräknar istället för varje tidssteg om det har ankommit en person eller inte sedan föregående tidssteg, vilket tillåter max en person per sekund. Ytterligare en felkälla är att pseudoslump använts, de femtio iterationerna bör dock ha reducerat den felkällan kraftigt.

De tidtagningar som gjordes under implementationen gav inga betydande skillnader. Felkällorna är många, alltifrån min egen dagsform till svåridentifierade buggar. Att den tidsstyrda modelleringen gick något fortare beror troligen på att delar av den händelsestyrda modellen kunde återanvändas.

Det är svårt att dra slutsatser om implementationssvårigheter då de problem som uppstod berodde oftare på rena logikfel som inte hade med paradigmen att göra. Dock visade sig händelsestyrning vara betydligt enklare att förstå, om än något svårt att komma in i. Dessutom resulterade händelsestyrning i färre rader kod och färre variabler än tidsstyrning.

Båda modellerna skulle mycket väl kunna utökas till att omfatta fler parametrar, fler hissar osv. Mest naturligt skulle det dock vara att fortsätta med den händelsestyrda modellen, då den är mycket mer lättarbetad. En utökning av den tidsstyrda modellen skulle sannolikt innebära ännu fler variabler att uppdatera och ta hänsyn till i varje tidssteg, något som lättare kunde undvikas i den händelsestyrda modellen.

Styrkan med tidsstyrd simulering är att kunna ändra tidssteget för att uppnå önskad noggrannhet. Denna möjlighet var starkt begränsad då tidssteget kunde förlängas endast under vissa villkor (se kapitel 2.4.2), beroende på systemets utpräglat diskreta egenskaper. Ett kortare tidsintervall hade heller inte förbättrat resultaten nämnvärt förutom att möjligen motverka skillnaderna i simuleringsresultaten (se ovan). Alltså finns inget egentligt skäl till att välja tidsstyrd simulering till en simulering av detta slag. Händelsestyrd simulering har styrkan att kunna ge ett exakt resultat för ett diskret system, varför händelsestyrning är det självklara valet i detta fall.

5. Referenser

5.1. Tryckt litteratur

Banks, Jerry; Carson, John S; Nelson, Barry L och Nicol, David M (2005). *Discrete-Event System Simulation, Fourth Edition*. ISBN 0-13-129342-7. New Jersey: Pearson Prentice Hall.

Blom, Gunnar; Enger, Jan; Englund, Gunnar; Grandell, Jan och Holst, Lars (2005). *Sannolikhetsteori och statistikteori med tillämpningar*. Lund: Studentlitteratur.

Ziegler, Bernard; Praehofer, Herbert och Gon Kim, Tag (2000). *Theory of Modeling and Simulation, Second Edition. Integrating Discrete Event and Continuous Complex Dynamic Systems*. ISBN 0-12-778455-1. San Diego: Academic Press.

5.2. Övriga källor

Carvalho, Marcio och Luna, Luis (2002). *Discrete and Continuous Simulation*. Hämtad 2012-02-19 från http://www.albany.edu/cpr/sdgroup/pad824/Discrete_and_Continuous_Simulation.ppt

Oracle (2011). *Class Random*. Hämtad 2012-04-11 från http://docs.oracle.com/javase/6/docs/api/java/util/Random.html

Sandblad, Bengt (n.d). *Kort om simulering och simulatorer*. Hämtad 2012-02-12 från http://www.it.uu.se/edu/course/homepage/opgui/vt03/Simulatordokumentation

Appendix

Den källkod som presenteras här är de två implementationer som gjorts samt kod för att testköra dessa implementationer. Koden är något formaterad för att passa i rapporten, men funktionaliteten är oförändrad.

A. Källkod för tidsstyrd simulering (TimeSim.java)

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.ArrayList;
import java.util.Random;

/**
 * The TimeSim class implements a time-slicing simulation
 * of an elevator in an office building. The simulation is
 * done as a part of the report "Simulering under press",
 * a bachelor thesis written in the course DD143X at the
 * Royal Institute of Technology (KTH) in Stockholm,
 * Sweden, Spring 2012.
 *
 * @author Markus Pettersson
 * @version 2012.04.11
 */

public class TimeSim implements ElevatorSimulation {

	private enum ElevatorMovement {
		UP, DOWN, AT_FLOOR
	};

	private enum ElevatorMode {
		UP, DOWN, IDLE
	}

	private class Person {
		public int timeStamp;
		public int destination;

		public Person(int timeStamp, int destination) {
			this.timeStamp = timeStamp;
			this.destination = destination;
		}

		@Override
		public String toString() {
			return "-> " + destination;
		}
	}

	private class Worker {
		private int timeLeft;

		public Worker(int timeLeft) {
			this.timeLeft = timeLeft;
		}

		public boolean tick() {
			timeLeft -= step;
			return timeLeft > 0;
		}
	}

	private class WorkPlace extends LinkedList<Worker> {
		private static final long serialVersionUID = 1L;

		@Override
		public String toString() {
			return Integer.toString(this.size());
```

```
		}
	}

	private class ElevatorQueue extends LinkedList<Person> {
		private static final long serialVersionUID = 1L;
	}

	private final int step, timePassingFloor, timeLoadingPassenger,
			numberOfFloors, maxCapacity, workTimeMeanValue,
			workTimeStdDeviation;

	private int numberOfEmployees;

	private Random random;
	private int currentTime;

	private ElevatorQueue[] personsInUpQueue, personsInDownQueue;
	private WorkPlace[] personsAtWork;
	private int[] personsInElevator;

	private int elevatorPosition;
	private ElevatorMovement movement;
	private ElevatorMode mode;

	private int loadingProgress;

	private boolean goToBoundaryFloor = false;

	private ArrayList<Integer> waitingTimes;
	private int maxWaitingTime;
	private String maxWaitingTimeInfo;

	public TimeSim(int numberOfFloors, int timeLoadingPassenger,
			int timePassingFloor, int maxCapacity, int workTimeHours, int wtStdDevMinutes)
{
		this.personsInUpQueue = new ElevatorQueue[numberOfFloors];
		this.personsInDownQueue = new ElevatorQueue[numberOfFloors];
		this.personsAtWork = new WorkPlace[numberOfFloors];
		for (int i = 0; i < numberOfFloors; i++) {
			personsInUpQueue[i] = new ElevatorQueue();
			personsInDownQueue[i] = new ElevatorQueue();
			personsAtWork[i] = new WorkPlace();
		}

		this.waitingTimes = new ArrayList<Integer>();

		this.personsInElevator = new int[numberOfFloors];

		this.step = 1;
		this.timeLoadingPassenger = timeLoadingPassenger;
		this.timePassingFloor = timePassingFloor;
		this.numberOfFloors = numberOfFloors;
		this.maxCapacity = maxCapacity;
		this.workTimeMeanValue = workTimeHours * 60 * 60;
		this.workTimeStdDeviation = wtStdDevMinutes * 60;

		elevatorPosition = 0;
		movement = ElevatorMovement.AT_FLOOR;
		mode = ElevatorMode.IDLE;

		this.random = new Random();

	}

	private void checkNewArrivals() {
		if (currentTime < 7 * 60 * 60 || currentTime > 10 * 60 * 60 - 1)
			return;

		if (random.nextDouble() < (double) step * (double) numberOfEmployees
				/ ((double) (3 * 60 * 60))) {
			personsInUpQueue[0].add(new Person(currentTime, 1 + random
					.nextInt(numberOfFloors - 1)));
			debugPrint("0 -> " + personsInUpQueue[0].getLast().destination);
		}

	}
```

```
private void updateWorkPlace(int floor) {
        Iterator<Worker> it = personsAtWork[floor].iterator();

        Worker w = null;
        while (it.hasNext()) {
                w = it.next();
                if (!w.tick()) { // time to go home
                        it.remove();
                        personsInDownQueue[floor].addLast(new Person(currentTime, 0));
                        debugPrint(floor + " -> 0");
                }
        }
}

private void updateElevator() {
        // check if the elevator is not idle anymore
        if (queuesEmpty() && elevatorEmpty()) {
                mode = ElevatorMode.IDLE;
                return;
        }

        if (mode == ElevatorMode.IDLE) {
                if (!elevatorEmpty()) // shouldn't be possible if elevator is idle.
                        throw new IllegalStateException(
                                        "Elevator is idle with people in it.");

                // so, someone has arrived to a queue and want's to go somewhere.
                // let's go to the closest of highestdownbutton and lowestupbutton
                int highestDown = highestDownButtonPressed(),
                        lowestUp = lowestUpButtonPressed();
                boolean downButtonCloser = Math.abs(highestDown - elevatorPosition) < Math
                                .abs(lowestUp - elevatorPosition);
                int targetFloor = 0;
                if ((lowestUp == -1 || downButtonCloser) && highestDown != -1) {
                        mode = ElevatorMode.DOWN;
                        targetFloor = highestDown;
                } else {
                        mode = ElevatorMode.UP;
                        targetFloor = lowestUp;
                }

                if (targetFloor == currentFloor())
                        movement = ElevatorMovement.AT_FLOOR;
                else
                        movement = targetFloor < currentFloor() ? ElevatorMovement.DOWN
                                        : ElevatorMovement.UP;

                goToBoundaryFloor = movement != ElevatorMovement.AT_FLOOR;
        }
        // move if the elevator is in movement
        switch (movement) {
        case AT_FLOOR:
                // unload or load people. mode can't be idle here since that will
                // return above...
                int currentFloor = currentFloor();
                if (mode == ElevatorMode.UP) {
                        loadOrUnload(personsInUpQueue[currentFloor],
                                        ElevatorMovement.UP);
                } else {
                        loadOrUnload(personsInDownQueue[currentFloor],
                                        ElevatorMovement.DOWN);
                }
                break;
        case UP:
                elevatorPosition += step;
                break;
        case DOWN:
                elevatorPosition -= step;
                break;
        }

        if (isOnFloor()) {
                if (goToBoundaryFloor) {
                        switch (mode) {
                        case UP:
```

```
                                if (currentFloor() == lowestUpButtonPressed()) {
                                        movement = ElevatorMovement.AT_FLOOR;
                                        goToBoundaryFloor = false;
                                        debugPrint("Is on lowest up floor");
                                }
                                break;
                        case DOWN:
                                if (currentFloor() == highestDownButtonPressed()) {
                                        movement = ElevatorMovement.AT_FLOOR;
                                        goToBoundaryFloor = false;
                                        debugPrint("Is on highest down floor");
                                }
                                break;
                        }
                } else { // stay if someone is heading here or from here
                        int currentFloor = currentFloor();
                        if (personsInElevator[currentFloor] > 0 ||
                                (mode == ElevatorMode.UP &&
                                        personsInUpQueue[currentFloor()].size() > 0 &&
                                        (maxCapacity - elevatorLoad()) > 0) ||
                                (mode == ElevatorMode.DOWN &&
                                        personsInDownQueue[currentFloor()].size() > 0 &&
                                        (maxCapacity - elevatorLoad()) > 0))
                                movement = ElevatorMovement.AT_FLOOR;
                }

        }

}

private void loadOrUnload(ElevatorQueue queue,
                ElevatorMovement previousDirection) {
        int currentFloor = currentFloor();
        if (personsInElevator[currentFloor] > 0
                        && loadingProgress() < personsInElevator[currentFloor]) {
                loadingProgress += step;
                if (loadingProgress() == personsInElevator[currentFloor]) {
                        // unloading is done
                        debugPrint("Unloaded " + personsInElevator[currentFloor]);
                        for (int i = 0; i < personsInElevator[currentFloor]; i++)
                                personsAtWork[currentFloor].add(new Worker(
                                                (int) (workTimeMeanValue + random.nextGaussian()
                                                        * workTimeStdDeviation)));
                        personsInElevator[currentFloor] = 0;
                        loadingProgress = 0;
                        if (queue.size() == 0 || elevatorLoad() == maxCapacity)
                                movement = getNextDirection(previousDirection);
                }
        } else { // it should be loading to do, otherwise the direction
                // should've been changed
                int loadSize = Math.min(queue.size(), maxCapacity - elevatorLoad());
                if (queue.size() > 0 && loadingProgress() < loadSize) {
                        loadingProgress += step;
                        if (loadingProgress() == loadSize) {
                                // loading is done
                                for (int i = 0; i < loadSize; i++) {
                                        personsInElevator[queue.getFirst().destination]++;
                                        waitingTimes.add(currentTime
                                                        - queue.removeFirst().timeStamp
                                                        - timeLoadingPassenger * (loadSize - i));
                                        if (waitingTimes.get(waitingTimes.size() - 1) >
                                                        maxWaitingTime) {
                                                maxWaitingTime = waitingTimes.get(waitingTimes
                                                                .size() - 1);
                                                maxWaitingTimeInfo = ts(currentTime / 3600) +
                                                        ":" + ts((currentTime / 60) % 60) + ":"
                                                        + ts(currentTime % 60) + ": floor "
                                                        + currentFloor;
                                        }
                                }
                                movement = previousDirection;
                                loadingProgress = 0;
                        }
                } else
                        throw new IllegalStateException(
                                        "No loading or unloading to do, but state"
```

```
                                                + " is AT_FLOOR");
        }
}

private ElevatorMovement getNextDirection(ElevatorMovement previousDirection) {
        int currentFloor = currentFloor();

        if (!elevatorEmpty()) { // move on in the same direction
                debugPrint("Return the same direction");
                return previousDirection;
        }

        if (queuesEmpty() && isOnFloor()) { // do nothing
                debugPrint("Set to idle");
                mode = ElevatorMode.IDLE;
                return ElevatorMovement.AT_FLOOR;
        }

        // elevator is empty, decide where to go
        if (mode == ElevatorMode.UP) {
                // only go up if there is work to do up there
                for (int i = currentFloor + 1; i < personsInUpQueue.length; i++)
                        if (personsInUpQueue[i].size() > 0)
                                return ElevatorMovement.UP;

                // if no one above want's to go up, go to highest down floor
                int highestDown = highestDownButtonPressed();
                if (highestDown != -1) {
                        mode = ElevatorMode.DOWN;
                        if (highestDown != currentFloor()) {
                                goToBoundaryFloor = true;
                                return highestDown < currentFloor ? ElevatorMovement.DOWN
                                                : ElevatorMovement.UP;
                        } else
                                return ElevatorMovement.AT_FLOOR;
                } else { // no down buttons pressed, go to up button
                        // the up button has to be under or at current floor
                        if (lowestUpButtonPressed() == currentFloor())
                                return ElevatorMovement.AT_FLOOR;
                        goToBoundaryFloor = true;
                        return ElevatorMovement.DOWN;
                }
        }

        if (mode == ElevatorMode.DOWN) {
                // go down if it's work to do there
                for (int i = 0; i < currentFloor; i++)
                        if (personsInDownQueue[i].size() > 0)
                                return ElevatorMovement.DOWN;

                // no work to do downwards, go to lowest up floor
                int lowestUp = lowestUpButtonPressed();
                if (lowestUp != -1) {
                        mode = ElevatorMode.UP;
                        if (lowestUp != currentFloor()) {
                                goToBoundaryFloor = true;
                                debugPrint("Go to boundary floor" + lowestUp);
                                return lowestUp < currentFloor ? ElevatorMovement.DOWN
                                                : ElevatorMovement.UP;
                        } else
                                return ElevatorMovement.AT_FLOOR;
                } else {
                        if (highestDownButtonPressed() == currentFloor())
                                return ElevatorMovement.AT_FLOOR;
                        goToBoundaryFloor = true;
                        return ElevatorMovement.UP;
                }
        }
        //this row shouldn't ever be run, but Java requires it
        return ElevatorMovement.AT_FLOOR;
}

private int loadingProgress() {
        return (loadingProgress / step) / timeLoadingPassenger;
}
```

```
private boolean queuesEmpty() {
        for (int i = 0; i < personsInUpQueue.length; i++)
                if (personsInUpQueue[i].size() > 0
                                || personsInDownQueue[i].size() > 0)
                        return false;
        return true;
}

private boolean elevatorEmpty() {
        for (int i = 0; i < personsInElevator.length; i++)
                if (personsInElevator[i] > 0)
                        return false;
        return true;
}

private int elevatorLoad() {
        int sum = 0;
        for (int i = 0; i < personsInElevator.length; i++)
                sum += personsInElevator[i];
        if (sum > maxCapacity)
                throw new IllegalStateException("Elevator is overfilled! (" + sum
                                + " people, max is " + maxCapacity + ")");
        return sum;

}

private int lowestUpButtonPressed() {
        for (int i = 0; i < personsInUpQueue.length; i++)
                if (personsInUpQueue[i].size() > 0)
                        return i;
        return -1;
}

private int highestDownButtonPressed() {
        for (int i = personsInDownQueue.length - 1; i >= 0; i--)
                if (personsInDownQueue[i].size() > 0)
                        return i;
        return -1;
}

private boolean isOnFloor() {
        return 0 == elevatorPosition % timePassingFloor;
}

private int currentFloor() {
        return (elevatorPosition / timePassingFloor) / step;
}

private void printTime() {
        System.out.print(ts(currentTime / 3600) + ":"
                        + ts((currentTime / 60) % 60) + ":" + ts(currentTime % 60));
}

private String ts(int value) {
        return value > 9 ? "" + value : "0" + value;
}

public int[] simulate(int numberOfEmployees) {
        this.numberOfEmployees = numberOfEmployees;

        currentTime = 7 * 3600;

        while (currentTime < 24 * 60 * 60 - 1) {
                checkNewArrivals();
                ElevatorMovement oldMove = movement;
                ElevatorMode oldMode = mode;
                updateElevator();
                if (oldMove != movement) { // print change
                        if (movement == ElevatorMovement.AT_FLOOR)
                                debugPrint("Arriving to " + currentFloor());
                        else
                                debugPrint("Leaving " + currentFloor());
                }
                if (oldMode != mode && mode == ElevatorMode.IDLE) {
                        debugPrint("(idle)");
                }
```

```
                        for (int i = 1; i < personsAtWork.length; i++)
                                updateWorkPlace(i);
                        currentTime += step;

                        if (currentTime % 3600 == 0)
                                debugPrint("Elevator position: " + currentFloor());
                }

                System.out.println("\n\n-------------------- RESULTS ------------------\n");

                int totalWaitingTime = 0, maxWaitingTime = 0;
                for (int i = 0; i < waitingTimes.size(); i++) {
                        totalWaitingTime += waitingTimes.get(i);
                        if (waitingTimes.get(i) > maxWaitingTime)
                                maxWaitingTime = waitingTimes.get(i);
                }

                if (waitingTimes.size() > 0) {
                        System.out.println("Average waiting time: "
                                        + (totalWaitingTime / waitingTimes.size() + " s"));
                        System.out
                                        .println("Maximum waiting time: " + maxWaitingTime + " s");
                        System.out.println(maxWaitingTimeInfo);
                } else
                        System.out.println("No waiting times recorded.");

                System.out.println("Peoplecount: " + waitingTimes.size() / 2);

                return new int[] {
                                waitingTimes.size() > 0 ? totalWaitingTime
                                                / waitingTimes.size() : 0, maxWaitingTime };
        }

        public static void main(String[] args) {
                new TimeSim(5, 5, 2, 5, 8, 30).simulate(100);
        }

        private static final boolean DEBUG = true;

        private void debugPrint(String text) {
                if (!DEBUG)
                        return;
                printTime();
                System.out.println(": " + text);
        }
}
```

B. Källkod för händelsestyrd simulering (ActionSim.java)

```
import java.util.Iterator;
import java.util.PriorityQueue;
import java.util.ArrayList;
import java.util.Random;
import java.util.LinkedList;

/**
 * The ActionSim class implements a next-event simulation
 * of an elevator in an office building. The simulation is
 * done as a part of the report "Simulering under press",
 * a bachelor thesis written in the course DD143X at the
 * Royal Institute of Technology (KTH) in Stockholm,
 * Sweden, Spring 2012.
 *
 * @author Markus Pettersson
 * @version 2012.04.11
 */

public class ActionSim implements ElevatorSimulation {
        public enum Type {
                ARRIVAL, DONE_LOADING, DONE_UNLOADING
        };

        public enum Direction {
                UP, DOWN
```

```
};

private class Event implements Comparable<Event> {
	private int timeStamp;

	public Event(int seconds) {
		this.timeStamp = seconds;
	}

	@Override
	public int compareTo(Event arg0) {
		return this.timeStamp - arg0.timeStamp();
	}

	public void printMe() {
		if (DEBUG)
			System.out.println(this.timeString() + " - " + this.toString());
	}

	public String timeString() {
		return ts(timeStamp / 3600) + ":" + ts((timeStamp / 60) % 60) + ":"
				+ ts(timeStamp % 60);
	}

	private String ts(int value) {
		return (value > 9 ? "" : "0") + value;
	}

	public int timeStamp() {
		return timeStamp;
	}
}

private class ElevatorEvent extends Event {
	public Type type;
	public int floor;

	public ElevatorEvent(int timestamp, Type type, int floor) {
		super(timestamp);
		this.type = type;
		this.floor = floor;
	}

	@Override
	public String toString() {
		if (type == Type.ARRIVAL) {
			return "Arriving at " + floor + "th floor";
		} else if (type == Type.DONE_LOADING) {
			return "Done loading at " + floor + "th floor";
		} else if (type == Type.DONE_UNLOADING) {
			return "Done unloading at " + floor + "th floor";
		}
		return "Type unrecognized in ElevatorEvent";
	}
}

private class PersonEvent extends Event {
	public int fromFloor, toFloor;

	public PersonEvent(int seconds, int fromFloor, int toFloor) {
		super(seconds);
		this.fromFloor = fromFloor;
		this.toFloor = toFloor;
	}

	@Override
	public String toString() {
		return "From " + fromFloor + " to " + toFloor;
	}
}

private class TimestampList extends LinkedList<Person> {
	private static final long serialVersionUID = 1L;

	@Override
	public String toString() {
```

```
                return Integer.toString(this.size());
        }
}

private class Person {
        public int timeStamp;
        public int destination;

        public Person(int timeStamp, int destination) {
                this.timeStamp = timeStamp;
                this.destination = destination;
        }

        @Override
        public String toString() {
                return "-> " + destination;
        }
}

private static final boolean DEBUG = true;

private final int numberOfFloors, workTimeMeanValue, workTimeStdDeviation,
        timeLoadPerson, timePassingFloor, elevatorMaxCapacity;

private PriorityQueue<Event> pq;
private Random rand;
private int[] personsOnFloor, personsInElevator;
private TimestampList[] personsInQueue;
private boolean[] upButtonPressed, downButtonPressed;

private boolean elevatorBusy;
private int restingPlace;
private Direction direction;

private ArrayList<Integer> waitingTimes;
private int maxWaitingTime;
private String maxWaitingTimeInfo;

public ActionSim(int numberOfFloors, int timeLoadPerson, int timePassingFloor,
                int elevatorMaxCapacity, int workTimeHours, int wtStdDevMinutes) {
        this.timeLoadPerson = timeLoadPerson;
        this.timePassingFloor = timePassingFloor;
        this.elevatorMaxCapacity = elevatorMaxCapacity;
        this.workTimeMeanValue = workTimeHours * 60 * 60;
        this.workTimeStdDeviation = wtStdDevMinutes * 60;

        pq = new PriorityQueue<Event>();
        rand = new Random();
        this.personsOnFloor = new int[numberOfFloors];
        this.personsInQueue = new TimestampList[numberOfFloors];
        for (int i = 0; i < personsInQueue.length; i++)
                personsInQueue[i] = new TimestampList();
        this.personsInElevator = new int[numberOfFloors];
        this.waitingTimes = new ArrayList<Integer>();
        this.numberOfFloors = numberOfFloors;
        this.direction = Direction.UP;

        this.upButtonPressed = new boolean[numberOfFloors];
        this.downButtonPressed = new boolean[numberOfFloors];
}

public int[] simulate(int numberOfWorkers) {
        addBuildingArrivals(numberOfWorkers);

        runElevatorSystem();

        if (DEBUG)
                System.out.println("\n---------------- RESULTS ----------------\n");

        int totalWaitingTime = 0, maxWaitingTime = 0, time;
        for (int i = 0; i < waitingTimes.size(); i++) {
                time = waitingTimes.get(i);
                totalWaitingTime += time;
                if (time > maxWaitingTime)
                        maxWaitingTime = time;
        }
```

```
        System.out.println("Average waiting: "
                        + (totalWaitingTime / waitingTimes.size()));
        System.out.println("Maximum waiting: " + maxWaitingTime + " ("
                        + maxWaitingTimeInfo + ")");

        System.out.println("People: " + waitingTimes.size() / 2);

        return new int[] { (totalWaitingTime / waitingTimes.size()),
                        maxWaitingTime };

}

private void runElevatorSystem() {
        Event nextEvent;
        while (!pq.isEmpty()) {
                nextEvent = pq.poll();

                if (nextEvent instanceof PersonEvent) {
                        //button is pressed
                        nextEvent.printMe();
                        PersonEvent p = (PersonEvent) nextEvent;
                        personsOnFloor[p.fromFloor]--;
                        personsInQueue[p.fromFloor].add(new Person(p.timeStamp(),
                                        p.toFloor));
                        if (p.fromFloor < p.toFloor)// going up
                                upButtonPressed[p.fromFloor] = true;
                        else
                                downButtonPressed[p.fromFloor] = true;

                        // set the elevator in movement if not busy
                        if (!elevatorBusy) {
                                elevatorBusy = true;
                                pq.add(new ElevatorEvent(p.timeStamp()
                                                + Math.abs(p.fromFloor - restingPlace)
                                                * timePassingFloor, Type.ARRIVAL, p.fromFloor));
                                direction = p.toFloor > p.fromFloor ? Direction.UP
                                                : Direction.DOWN;
                        }
                } else if (nextEvent instanceof ElevatorEvent) {
                        ElevatorEvent e = (ElevatorEvent) nextEvent;
                        if (e.type == Type.ARRIVAL) {
                                if (personsInQueue[e.floor].size() > 0
                                                || personsInElevator[e.floor] > 0)
                                        e.printMe();
                                // unload all people heading for this floor
                                pq.add(new ElevatorEvent(e.timeStamp()
                                                + personsInElevator[e.floor] * timeLoadPerson,
                                                Type.DONE_UNLOADING, e.floor));
                                personsOnFloor[e.floor] += personsInElevator[e.floor];
                                if (e.floor != 0) { // arriving to workplace, add going home
                                        // event
                                        for (int i = 0; i < personsInElevator[e.floor]; i++) {
                                                pq.add(new PersonEvent(e.timeStamp() + i
                                                                * timeLoadPerson + getWorkTime(),
                                                                e.floor, 0));
                                        }
                                }
                                personsInElevator[e.floor] = 0;
                        }
                        if (e.type == Type.DONE_UNLOADING) {
                                loadPeople(e);
                        }
                        if (e.type == Type.DONE_LOADING) {
                                // if more people have arrived to the queue since the last
                                // time, load them too if possible
                                if (personsInQueue[e.floor].size() > 0
                                                && elevatorMaxCapacity >
                                                numberOfPeopleInElevator()) {
                                        loadPeople(e);
                                        continue;
                                }
                                // else we're done loading people, go to next floor

                                if (elevatorIsEmpty() && noButtonsPressed()) {
                                        elevatorBusy = false; // do nothing more
                                        restingPlace = e.floor;
```

```
                                        continue;
                                }
                                // find out where to go next
                                int nextFloor = -1;
                                if (direction == Direction.UP) {
                                        if (!elevatorIsEmpty()
                                                        || highestUpButtonPressed() > e.floor)
                                                nextFloor = e.floor + 1;
                                        else { // go to highest "down floor"
                                                nextFloor = highestDownButtonPressed();
                                                if (nextFloor != -1)
                                                        direction = Direction.DOWN;
                                                else
                                                        // the pressed button is an up button
                                                        nextFloor = lowestUpButtonPressed();
                                        }
                                } else if (direction == Direction.DOWN) {
                                        if (!elevatorIsEmpty()
                                                        || lowestDownButtonPressed() < e.floor)
                                                nextFloor = e.floor - 1;
                                        else { // go to lowest "up floor"
                                                nextFloor = lowestUpButtonPressed();
                                                if (nextFloor != numberOfFloors)
                                                        direction = Direction.UP;
                                                else
                                                        nextFloor = highestDownButtonPressed();
                                        }
                                }
                                pq.add(new ElevatorEvent(e.timeStamp()
                                                + Math.abs(e.floor - nextFloor) *
                                                timePassingFloor, Type.ARRIVAL, nextFloor));
                                if (personsInQueue[e.floor].size() > 0) {
                                        if (direction == Direction.UP)
                                                upButtonPressed[e.floor] = true;
                                        else
                                                downButtonPressed[e.floor] = true;
                                }
                        }
                } else {
                        System.out.println("Unexpected event type.");
                        return;
                }
        }
}

private void loadPeople(ElevatorEvent e) {
        // how many persons can be loaded?
        int queueSize = countQueuePeople(e.floor, direction);
        int pCount = Math.min(elevatorMaxCapacity - numberOfPeopleInElevator(),
                        queueSize);

        if (pCount > 0 && DEBUG){
                System.out.println(e.timeString() + " - loading " + pCount
                                + " people.");
        }

        Person person;
        int loadedPersons = 0;
        Iterator<Person> it = personsInQueue[e.floor].iterator();
        while (it.hasNext() && numberOfPeopleInElevator() < elevatorMaxCapacity) {
                person = it.next();
                if ((direction == Direction.UP && person.destination < e.floor)
                                || (direction == Direction.DOWN && person.destination >
                                e.floor))
                        continue;
                it.remove();
                int time = e.timeStamp() - person.timeStamp + loadedPersons
                                * timeLoadPerson;
                waitingTimes.add(time);
                //check if this time is a new maximum
                if (time > maxWaitingTime) {
                        maxWaitingTime = time;
                        maxWaitingTimeInfo = e.timeString() + ": waited on floor "
                                        + e.floor;
                }
                loadedPersons++;
```

```
                personsInElevator[person.destination]++;
        }
        pq.add(new ElevatorEvent(e.timeStamp() + pCount * timeLoadPerson,
                        Type.DONE_LOADING, e.floor));

        if (direction == Direction.UP)
                upButtonPressed[e.floor] = false;
        if (direction == Direction.DOWN)
                downButtonPressed[e.floor] = false;
}

private int countQueuePeople(int floor, Direction direction) {
        int count = 0;
        for (Person person : personsInQueue[floor])
                if ((direction == Direction.UP && person.destination > floor)
                                || (direction == Direction.DOWN && person.destination < floor))
                        count++;
        return count;
}

private void addBuildingArrivals(int numberOfWorkers) {
        double time = 0;
        double timeSpan = 3 * 60 * 60;
        while (time < timeSpan){
                time += - Math.log(rand.nextDouble()) / (numberOfWorkers / timeSpan);
                personsOnFloor[0]++;
                pq.add(new PersonEvent(7 * 60 * 60 + (int)time, 0, 1 + rand
                                .nextInt(numberOfFloors - 1)));
        }
}

private int getWorkTime() {
        return (int) (workTimeMeanValue + workTimeStdDeviation * rand.nextGaussian());
}

private int numberOfPeopleInElevator() {
        int ret = 0;
        for (int n : personsInElevator)
                ret += n;
        return ret;
}

private boolean elevatorIsEmpty() {
        for (int n : personsInElevator)
                if (n > 0)
                        return false;
        return true;
}

private boolean noButtonsPressed() {
        for (int i = 0; i < numberOfFloors; i++)
                if (upButtonPressed[i] || downButtonPressed[i])
                        return false;
        return true;
}

private int highestDownButtonPressed() {
        for (int i = numberOfFloors - 1; i >= 0; i--)
                if (downButtonPressed[i])
                        return i;
        return -1;
}

private int lowestDownButtonPressed() {
        for (int i = 0; i < numberOfFloors; i++)
                if (downButtonPressed[i])
                        return i;
        return numberOfFloors;
}

private int highestUpButtonPressed() {
        for (int i = numberOfFloors - 1; i >= 0; i--)
                if (upButtonPressed[i])
                        return i;
        return -1;
}
```

```
	private int lowestUpButtonPressed() {
		for (int i = 0; i < numberOfFloors; i++)
			if (upButtonPressed[i])
				return i;
		return numberOfFloors;
	}

	public static void main(String args[]) {
		new ActionSim(5, 5, 2, 5, 8, 30).simulate(100);
	}
}
```

C. Källkod för testkörning (MultiSimulation.java, ElevatorSimulation.java)

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.util.Arrays;

/**
 * The MultiSimulation class implements test simulation
 * functionality of two implementation fo the same model:
 * an elevator in an office building. The simulation is
 * done as a part of the report "Simulering under press",
 * a bachelor thesis written in the course DD143X at the
 * Royal Institute of Technology (KTH) in Stockholm,
 * Sweden, Spring 2012.
 *
 * @author Markus Pettersson
 * @version 2012.04.11
 */

public class MultiSimulation {

	private static PrintStream oldOut;
	private static final int compareIterations = 50;
	/**
	 * @param args
	 */
	public static void main(String[] args) {
		oldOut = System.out;
		try {
			System.setOut(new PrintStream(
					new FileOutputStream("D:\\simulationoutput.txt")));
		} catch (FileNotFoundException e) {
			return;
		}

		println("Class \t\taverage / max time");

		// compare(new XSim(a, b, c, d, e, f), g) runs a simulation with the following
		// parameters:
		// a = number of floors
		// b = loadingtime/passenger
		// c = speed (seconds passing one floor)
		// d = capacity
		// e = average work time (in hours)
		// f = standard deviation of work time (in minutes)
		// g = number of employees
		simulate(new TimeSim(2, 10, 10, 1, 8, 0), 1);
		simulate(new ActionSim(2, 10, 10, 1, 8, 0), 1);

		simulate(new TimeSim(5, 10, 15, 3, 8, 30), 50);
		simulate(new ActionSim(5, 10, 15, 3, 8, 30), 50);

		simulate(new TimeSim(10, 2, 5, 4, 8, 30), 200);
		simulate(new ActionSim(10, 2, 5, 4, 8, 30), 200);

		simulate(new TimeSim(20, 1, 2, 15, 8, 60), 1500);
		simulate(new ActionSim(20, 1, 2, 15, 8, 60), 1500);

		System.setOut(oldOut);
	}
```

```
	private static void simulate(ElevatorSimulation sim, int employeeCount){

		int[] maxTimes = new int[compareIterations],
			averageTimes = new int[compareIterations];

		int[] result;
		for (int i = 0; i < compareIterations; i++){
			result = sim.simulate(employeeCount);
			averageTimes[i] = result[0];
			maxTimes[i] = result[1];
		}

		//sort to find median value
		Arrays.sort(averageTimes);
		Arrays.sort(maxTimes);

		println((sim instanceof TimeSim ? "TimeSim" : "ActionSim") + ": \t" +
				averageTimes[compareIterations / 2] + " / " +
				maxTimes[compareIterations / 2]);
	}

	private static void println(String text){
		oldOut.println(text);
	}

}

/**
 * Interface used for easier test runs. The interface
 * is a part of the report "Simulering under press", a
 * bachelor thesis written in the course DD143X at the
 * Royal Institute of Technology (KTH) in Stockholm,
 * Sweden, Spring 2012.
 *
 * @author Markus Pettersson
 * @version 2012.04.11
 */

public interface ElevatorSimulation {
	/**
	 * Running a simulation of an elevator.
	 * @return Array containing two values: average waiting time and maximum waiting time
	 */
	public int[] simulate(int numberOfEmployees);
}
```