

ContactCenters Simulation Library User's Guide

Examples of simulation programs

Version: March 4, 2014

ERIC BUIST

We present examples of Java programs based on the ContactCenters library. The examples are commented and are an excellent starting point for learning how to construct new contact center simulators or extend existing ones.

Contents

Overview	2
1 $M/M/c$ queue and variants	4
1.1 $M/M/c$ queue	4
1.2 Supporting abandonment	11
1.3 Clearing the queue at the end of the day	16
1.4 Limiting the queue capacity	19
1.5 Preparing for parallel simulations	21
2 A simplified call center with multiple periods	26
3 A simple multi-skill contact center	34
3.1 Implementing the model	34
3.2 Adding a contact-by-contact trace	40
3.3 Rerouting queued contacts	44
4 Telethon call center	51
4.1 Implementing the model	51
4.2 Modeling the message delay	59
4.3 Correcting the estimator for occupancy ratio	61
5 Bilingual contact center	65
6 Bank model	77
7 Teamwork contact center	94
8 Blend call center	111

Overview

The ContactCenters library provides some generic precompiled simulators for the most common cases. These simulators use XML parameter files and can estimate several performance measures. However, they do not cover every contact center and it is sometimes needed to make customizations to match specific needs. In addition, a simulator specialized for the needed model is often more efficient than a generic system adapted for a wider range of contact centers.

The ContactCenters library is more than a set of programs; it is a toolkit permitting the construction of simulators. It provides building blocks to model the simulation logic of contact centers using the SSJ simulation library [8] and Colt [6]. The simulated horizon can be finite or infinite.

We present some Java simulation programs using the ContactCenters library. These examples will help the reader better understand how the contact center simulation works and how new models can be implemented by combining or extending the components provided by the library. Reading these examples is a good way to start learning about the ContactCenters library. While studying the programs, the reader can refer to the functional definitions of ContactCenters classes and methods in the guides of the corresponding packages. It is preferable to refer to the PDF versions of the guides, because they contain a more detailed and complete documentation than the HTML versions, which are better suited for quick on-line referencing for those who are already familiar with ContactCenters.

A basic understanding of the SSJ library is needed to read these examples. See the SSJ examples and API documentation for more information [8]. Knowledge of the Java programming language is also strongly recommended. See [3] or the Java tutorial and Java API documentation for more information. Familiarity with contact center simulation is also recommended [5, 9].

Section 1 presents a $M/M/c$ queue with FIFO discipline to illustrate the structure of a basic ContactCenters simulator. The simple system is then extended several ways to demonstrate various aspects of the library. Section 2 gives a second simplified call center supporting multiple periods. Section 3 demonstrates how to construct a non-stationary multi-skill contact center. Sections 4, 5, 6, and 7 describe four examples of non-stationary simulators which have been adapted from Rockwell's *Arena Contact Center Edition* sample models in order to help the user map the features of this commercial tool to the ContactCenters library. The last example, in section 8, models a blend contact center in order to demonstrate how to put a dialer into action.

Since these examples model call centers, a contact corresponds to a phone call in this guide. However, the library is capable of simulating any type of contacts such as fax, e-mail, chat, etc.

The examples do not implement the high-level interface defined in the `app` package for simplicity. In most cases, it is recommended to implement it since it allows other programs such as statistical analyzers or optimizers to better interact with the simulator.

Moreover, most examples shown here use the basic SSJ API which does not support parallel simulations. This means that if several instances of the example classes are created in a Java program, and one instance is associated with one thread, the resulting multi-threaded application will not work, because all instances will share the same simulation clock and event list. Making parallel simulations requires the use of the `Simulator` class rather than static methods in the `Sim` class. However, multiple instances of the `java` command executing copies of the following examples can run concurrently without problem.

1 *M/M/c* queue and variants

1.1 *M/M/c* queue

One of the simplest system that can be modeled with the library is a call center with a single call type, and a single agent group having a fixed capacity c , i.e., a *M/M/c* queue with FIFO discipline. This model is implemented for demonstration purposes only since simple formulas are available to compute the average waiting time, the average queue length, etc. [7] It could also be implemented without ContactCenters, as shown in the Examples user's guide of SSJ [8].

Arrivals follow an homogeneous Poisson process with rate λ , and customers are queued if they cannot be served immediately. Abandonment is not allowed and service times are i.i.d. exponential variables with mean $1/\mu$.

The system estimates the service level, the expected speed of answer, and the agents' occupancy ratio. Let $S_G(s)$ be the number of served contacts having waited in queue for less than an *acceptable waiting time* s , and let S be the total number of served contacts. Since no abandonment occurs, this is equivalent to the number of arrivals. Let W_S be the sum of waiting times for served contacts. The *service level* is defined by

$$g_0(s) = \frac{\mathbb{E}[S_G(s)]}{\mathbb{E}[S]}. \quad (1)$$

The *speed of answer* is defined by

$$w_X = \frac{\mathbb{E}[W_S]}{\mathbb{E}[S]}. \quad (2)$$

The *agents' occupancy ratio* is defined by

$$o = \frac{\mathbb{E} \left[\int_0^{t_P} N_B(t) dt \right]}{\mathbb{E} \left[\int_0^{t_P} (N(t) + N_G(t)) dt \right]}. \quad (3)$$

t_P corresponds to the time at which the contact center closes. $N_G(t)$ gives the number of *ghost agents*, i.e., agents that must exit the system after finishing their current service. Since ghost agents appear only when $N(t)$ changes with time, $N_G(t) = 0$ for this model. To get an estimate of each performance measure of interest, we perform n independent replications of a simulation on a finite horizon.

Most elements of the contact center simulator are provided by the ContactCenters library as independent components. The modeler's task consists of assembling these elements and providing some glue code for statistical collecting, routing and stopping conditions, some elements being optional. For this example, whose code is presented on Listing 1, simpler programs can be written, especially if the library is not used. However, we prefer to demonstrate how to balance simplicity and scalability in a contact center simulation.

The first part of the program defines and constructs various objects while the second part contains the simulation logic. The `main` method, at the end of the file, constructs a

$M/M/c$ queue simulator, triggers a simulation by calling `simulate`, and prints a statistical report using `printStatistics`. A `Chrono` is used to determine the CPU time taken by the simulation. The `simulate` method calls `simulateOneDay` n times to perform the replications while `simulateOneDay` initializes the system, performs the simulation, and collects some observations.

Listing 1: A $M/M/c$ queue implemented with `ContactCenters`

```
import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.contact.ContactArrivalProcess;
import umontreal.iro.lecuyer.contactcenters.contact.ContactFactory;
import umontreal.iro.lecuyer.contactcenters.contact.PoissonArrivalProcess;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.queue.StandardWaitingQueue;
import umontreal.iro.lecuyer.contactcenters.queue.WaitingQueue;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.contactcenters.router.QueuePriorityRouter;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.server.AgentGroup;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;
import umontreal.iro.lecuyer.contactcenters.server.GroupVolumeStat;
import umontreal.iro.lecuyer.randvar.ExponentialGen;
import umontreal.iro.lecuyer.rng.MRG32k3a;
import umontreal.iro.lecuyer.simevents.Event;
import umontreal.iro.lecuyer.simevents.Sim;
import umontreal.iro.lecuyer.stat.FunctionOfMultipleMeansTally;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.util.Chrono;
import umontreal.iro.lecuyer.util.RatioFunction;

public class MMC {
    // Input data (times are in minutes)
    static final double LAMBDA      = 4.2;      // Arrival rate
    static final double MU         = 0.5;      // Service rate
    static final double AWT        = 20/60.0;  // Acceptable waiting time
    static final int NUMAGENTS     = 12;
    static final int[][] TYPETOGROUPMAP = { { 0 } };
    static final int[][] GROUPTOTYPEMAP = { { 0 } };
    static final double LEVEL      = 0.95;      // Level of conf. int.
    static final double DAYLENGTH  = 8.0*60.0; // Eight hours
    static final int NUMDAYS       = 1000;

    // Contact center components
    ContactArrivalProcess arrivProc;
    AgentGroup agents;
    WaitingQueue queue;
    Router router;
```

```

// Service time generator
ExponentialGen sgen = new ExponentialGen (new MRG32k3a(), MU);

// Counters and probe used during replications
int numGoodSL, numServed;
double sumWaitingTimes;
GroupVolumeStat vstat;
double Nb, N;

// Collectors with one obs./rep.
Tally goodSL = new Tally ("Number of contacts in target");
FunctionOfMultipleMeansTally serviceLevel = new FunctionOfMultipleMeansTally
    (new RatioFunction(), "Service level", 2),
    occupancy = new FunctionOfMultipleMeansTally
    (new RatioFunction(), "Occupancy ratio", 2),
    speedOfAnswer = new FunctionOfMultipleMeansTally
    (new RatioFunction(), "Speed of answer", 2);

MMC() {
    arrivProc = new PoissonArrivalProcess
        (new MyContactFactory(), LAMBDA, new MRG32k3a());
    agents = new AgentGroup (NUMAGENTS);
    queue = new StandardWaitingQueue();
    router = new QueuePriorityRouter (TYPETOGROUPMAP, GROUPTOTYPEMAP);
    arrivProc.addNewContactListener (router);
    router.setAgentGroup (0, agents);
    router.setWaitingQueue (0, queue);
    router.addExitedContactListener (new MyContactMeasures());
    vstat = new GroupVolumeStat (agents);
}

// Creates the new contacts
class MyContactFactory implements ContactFactory {
    public Contact newInstance() {
        final Contact contact = new Contact();
        contact.setDefaultServiceTime (sgen.nextDouble());
        return contact;
    }
}

// Updates counters when a contact exits
class MyContactMeasures implements ExitedContactListener {
    public void blocked (Router router, Contact contact, int bType) {}
    public void dequeued (Router router, DequeueEvent ev) {}
    public void served (Router router, EndServiceEvent ev) {
        ++numServed;
    }
}

```

```

        final double qt = ev.getContact().getTotalQueueTime();
        if (qt < AWT) ++numGoodSL;
        sumWaitingTimes += qt;
    }
}

class EndSimEvent extends Event {
    @Override
    public void actions() { endSim(); }
}

void endSim() {
    arrivProc.stop();
    Nb = vstat.getStatNumBusyAgents().sum();    // Int. for N_b0(t)
    N = vstat.getStatNumAgents().sum();         // Int. for N_0(t)
}

void simulateOneDay() {
    Sim.init();    // Initialize clock and clear event list
    new EndSimEvent().schedule (DAYLENGTH);
    arrivProc.init();    agents.init();    queue.init();
    numServed = numGoodSL = 0;
    sumWaitingTimes = 0;
    vstat.init();
    arrivProc.start();
    Sim.start();    // Simulation runs here
    addObs();
}

void addObs() {
    goodSL.add (numGoodSL);
    serviceLevel.add (numGoodSL, numServed);
    speedOfAnswer.add (sumWaitingTimes, numServed);
    occupancy.add (Nb, N);
}

void simulate (int days) {
    goodSL.init();
    serviceLevel.init();
    occupancy.init();
    for (int r = 0; r < days; r++) simulateOneDay();
}

public void printStatistics() {
    System.out.println (goodSL.reportAndCISTudent (LEVEL, 3));
    System.out.println (serviceLevel.reportAndCIDelta (LEVEL, 3));
    System.out.println (speedOfAnswer.reportAndCIDelta (LEVEL, 3));
}

```



```

        System.out.println (occupancy.reportAndCIDelta (LEVEL, 3));
    }

    public static void main (String[] args) {
        final MMC s = new MMC();
        final Chrono timer = new Chrono();
        s.simulate (NUMDAYS);
        System.out.println ("CPU time: " + timer.format());
        s.printStatistics();
    }
}

```

At the beginning of the program, the necessary classes are retrieved using `import` statements, and a class named `MMC` and representing the simulator is declared. Because variants of this example will be presented later as subclasses of `MMC`, some indirections that might appear unnecessary will be used in the program.

In this example and most of the following ones, hard-coded constants are used as input data for simplicity. In general, it is strongly recommended to use a parameter file, since it allows modification of the parameters without recompiling the simulator. The package `umontreal.iro.lecuyer.xmlconfig` provides facilities to support XML-based configuration files. As with the underlying SSJ library, the simulation logic of `ContactCenters` makes no assumption about the time unit, but input data should be defined consistently to avoid confusion in the future. In our examples, unless specified otherwise, the time unit is the minute. For example, the arrival rate specifies the expected number of contacts per minute.

The objects representing components of the contact center, such as the arrival process, waiting queue, agent group, as well as the random variate generator for the service times can be accessed through fields of the `MMC` class. Since the components themselves do not compute any statistic, the simulator defines some counters and statistical collectors to estimate the performance measures of interest. Counters are used during replications to compute observations while statistical collectors are used at the end to collect them. We use integers for counting events rather than statistical probes to clearly show the difference between the two steps during the simulation. `vstat` is used to compute the integrals of the number of agents over simulation time, for estimating the occupancy ratio. Each agent group volume statistical object contains accumulates observing the total number of agents, the number of busy agents, etc. for a single agent group. For each counter, a statistical probe is defined to collect the observations computed during each replication.

When instantiating the `MMC` class, the constructor, shown after the declaration of fields, creates the contact center objects and connects them together. Inbound contacts are simulated using an arrival process which needs to determine how to construct contacts. Contacts are represented by `Contact` instances which are constructed by a *factory*, i.e., a mechanism permitting the instantiation of objects from the `Contact` class, or any subclass. In contrast, a constructor instantiates objects from a single class. A contact factory is defined by creating a class implementing the `ContactFactory` interface. The `newInstance` method, specified by

this interface, must construct and return an initialized instance of **Contact** (or any subclass of **Contact**). In this example, **MyContactFactory** creates the contacts and sets their service times.

Arrival processes of different types could use the same contact factory, but in this example, a single Poisson process is needed. In addition to the contact factory, the process requires a random stream for generating the exponential inter-arrival times, and an arrival rate.

Constructing the agent group only requires the number of agents in it. This object manages the service of contacts and keeps counters for the number of free and busy agents. All agents of a group are usually considered identical, simplifying the routing significantly. The agent group automatically schedules events to manage service termination if service times are associated with handled contacts.

The constructed standard waiting queue orders waiting contacts based on their arrival times only, implementing *First In First Out* (FIFO) or *Last In First Out* (LIFO) disciplines; the choice between FIFO and LIFO is made by the routing policy.

The router uses a queue priority policy, but in fact, any routing policy will act identically here since there is a single call type and a single agent group. The only important parameters are the type-to-group and group-to-type maps which are always the same for a single-type and single-group contact center. To maximize the flexibility of the library, no assumption is made about the connections between components. Arrival processes, agent groups, and waiting queues must be manually connected to the router.

The components of the library communicate using the *observers* design pattern [4]. In that setting, an *observable* object, also called a *broadcaster*, is capable of transmitting some information to a list of registered listeners known at runtime only. A *listener*, also called an *observer*, is an object receiving information broadcast by an observable object. In Java, listeners are required to implement a particular interface the broadcaster uses to transmit the information through specified methods. Each component of the library defines its own listener interface to avoid the necessity of type casting by observers.

Each time a new contact is created by the factory and returned to the arrival process, it is broadcast to a list of registered *new-contact listeners*. Such a listener can correspond to any object implementing the **NewContactListener** interface, which specifies a **newContact** method receiving **Contact** objects. For the contacts to be processed, the list bound to the arrival process being used must include a reference to the router, which can listen to new contacts.

While it does not interact with the arrival process, the router affects the waiting queue by inserting new contacts when they cannot be served immediately, or by pulling ones when agents are free. It also affects agent groups by sending them contacts to serve. The linking mechanism is different from arrival processes to emphasize the two types of relationships.

The simulator also needs to be notified about contacts leaving the router through an *exited-contact listener*. Such a listener object implements an interface called **ExitedContactListener** specifying methods to receive exited contacts. Each method transmits all available information, permitting the simulator to count events under complex conditions.

These connections can easily be changed during the life cycle of the simulator: new listeners can be added, and old ones can be removed. It is even possible, with some care, to completely replace the router at runtime, allowing different routing policies to be tested without recreating the entire system.

The `simulate` method initializes the statistical collectors and performs n independent replications by using `simulateOneDay`. The latter method may be considered as the heart of the program. First, the simulation clock is initialized, and the event list is cleared by `Sim.init`. An event for the end of the day is then scheduled, and all the contact center's components are reset, avoiding any side effect from previous replications. This initialization disables the arrival process. Then, after the event counters are reset to 0, the arrival process is started, scheduling the first contact. The simulation can now be started using `Sim.start`. This instructs SSJ to start executing events, until the event list is empty.

When the first arrival occurs, the Poisson arrival process calls the `newInstance` method of `MyContactFactory` to get a new contact object. This method simply constructs the appropriate object and generates a service time which is stored in the contact object to be retrieved later by the agent group. After the new contact is returned, it is broadcast to the router, and a new arrival is scheduled. The user only needs to provide the `newInstance` method, the rest of the logic being implemented once in `ContactArrivalProcess`, and inherited by `PoissonArrivalProcess`.

In its `newContact` method, the router sends the newly-constructed contact to an agent, or adds it to the waiting queue if no agent is available. The way the agent or the waiting queue are selected, called *agent selection* and *queue selection*, respectively, depends on the algorithm implemented in the chosen subclass of `Router`. In this example, when a service starts, a free agent becomes busy, and an event is scheduled to happen after the service time elapses. By default, the service time is extracted from contacts, but this can be customized in many ways if needed, as we will do in section 6. Since the default service time is infinite, it is important to set a service time in the contact factory, or change the service time generator of the agent group. If the contact is queued, the contact is added, with extra information, to an internal linked list.

When a service ends, the router is notified, the contact exits the system, and it is broadcast to the `served` method of `MyContactMeasures`. This method receives all information about the served contact through the *end-service event* object. In this example, after a service is counted, the waiting time of the contact must be obtained: the served `Contact` is extracted from the end-service event, and the total queue time is queried. Note that this obtains the *cumulative* queue time, i.e., the time spent by the contact in all waiting queues. In simple systems, this is the same as the waiting time in the last queue since the contact waits only once. Other information such as the starting time of the service, and the reason why the service ended can be extracted from the event. The two other methods of `MyContactMeasures` contain no code, because no contact is blocked or abandons in this model.

When an agent became free due to the service termination, the router tries to perform *contact selection*, i.e., it scans waiting queues to assign a waiting contact to the free agent. Without contact selection, all queued contacts would wait forever. In this example, the

router simply takes the first contact in the waiting queue, implementing a FIFO discipline. If the queue is empty at pull time, the affected agent remains free until the following arrival.

When the simulation clock reaches `DAYLENGTH`, the `EndSimEvent` is executed, which calls `endSim` to disable the arrival process. This cancels the last scheduled arrival, breaking the life cycle of the process. If the arrival process is never disabled, the simulation will run forever, and results will never be printed. `endSim` also stores the integrals of $N(t)$ and $N_B(t)$ for the occupancy ratio to exclude the wrap-up period following the closing time. No more arrival occurs, but in-progress and queued contacts are served before the end of the replication. After all these services are over, `Sim.start` ends and the `simulateOneDay` method continues its execution. Note that in a further example, the `endSim` method will be overridden to change how the simulation is terminated.

It is important not to end the simulation abruptly using `Sim.stop`, otherwise the system will end non-empty: queues will contain contacts, and some agents will still be serving customers. If the simulation is stopped abruptly, the program needs to take this non-empty state into account during statistical collecting.

The `addObs` method is called to add observations to statistical collectors. In further examples, this will be overridden to change how observations are collected. The number of served contacts as well as the service level for the replication are added into tallies. A tally for functions of multiple means is used for the service level to get a ratio of averages instead of an average of ratios. This estimates the long-term service level rather than the performance measure during a single day. The integrals for $N_B(t)$ and $N(t)$ are added to tallies for estimating the occupancy ratio. As with the service level, we try to estimate a long-term measure.

After n independent replications of this process, we have statistical results for the estimated performance measures, including sample averages, sample variances, and Student- t confidence intervals. For the ratios, confidence intervals are computed using the delta theorem [10]. The `printStatistics` method is used to show a statistical report similar to Listing 2.

1.2 Supporting abandonment

To support abandonment, some modifications to the previous program are necessary: patience times must be generated and associated with contacts, and the contacts having abandoned need to be counted. Even if the abandonment count is not required, the service level computation must be altered as follows to take this new aspect into account.

Let $L_B(s)$ be the number of contacts having abandoned after a waiting time greater than s , and let L be the total number of abandoned contacts. The new service level estimator is defined by

$$g_1(s) = \frac{\mathbb{E}[S_G(s)]}{\mathbb{E}[S + L_B(s)]}. \quad (4)$$

If abandonment is disabled, the new estimator reverts to (1).

Listing 2: Results of the program MMC

```

CPU time: 0:0:1.56
REPORT on Tally stat. collector ==> Number of contacts in target
  num. obs.      min      max      average      standard dev.
    1000    1588.000    1964.000    1816.158      57.043
95.0% confidence interval for mean (student): ( 1812.618, 1819.698 )

REPORT on Tally stat. collector ==> Service level
  func. of averages      standard dev.      num. obs.
        0.902          0.033          1000
95.0% confidence interval for function of means: (    0.900,    0.904 )

REPORT on Tally stat. collector ==> Speed of answer
  func. of averages      standard dev.      num. obs.
        0.099          0.045          1000
95.0% confidence interval for function of means: (    0.097,    0.102 )

REPORT on Tally stat. collector ==> Occupancy ratio
  func. of averages      standard dev.      num. obs.
        0.695          0.023          1000
95.0% confidence interval for function of means: (    0.694,    0.697 )

```

Listing 3 presents an extension of the previous program supporting abandonment. Instead of rewriting the whole simulator, we inherit from it and change its behavior by overriding appropriate methods.

Listing 3: An extension of the *M/M/c* model supporting abandonment

```

import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;
import umontreal.iro.lecuyer.randvar.ExponentialGen;
import umontreal.iro.lecuyer.rng.MRG32k3a;
import umontreal.iro.lecuyer.stat.FunctionOfMultipleMeansTally;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.util.Chrono;
import umontreal.iro.lecuyer.util.RatioFunction;

public class MMCAb extends MMC {
    static final double NU = 1.0;      // 1/Mean patience time

```

```

ExponentialGen pgen = new ExponentialGen (new MRG32k3a(), NU);
int numAbandoned;
int numAbandonedAfterAWT;
Tally abandoned = new Tally ("Number of contacts having abandoned");
FunctionOfMultipleMeansTally serviceLevel = new FunctionOfMultipleMeansTally
    (new RatioFunction(), "Service level with abandonment", 2);

MMCAb() {
    super();
    arrivProc.setContactFactory (new MyContactFactoryAb());
    router.addExitedContactListener (new MyContactMeasuresAb());
}

class MyContactFactoryAb extends MyContactFactory {
    @Override
    public Contact newInstance() {
        final Contact contact = super.newInstance();
        contact.setDefaultPatienceTime (pgen.nextDouble());
        return contact;
    }
}

class MyContactMeasuresAb implements ExitedContactListener {
    public void blocked (Router router, Contact contact, int bType) {}
    public void dequeued (Router router, DequeueEvent ev) {
        ++numAbandoned;
        if (ev.getContact().getTotalQueueTime() >= AWT) ++numAbandonedAfterAWT;
    }
    public void served (Router router, EndServiceEvent ev) {}
}

@Override
void simulateOneDay() {
    numAbandoned = numAbandonedAfterAWT = 0;
    super.simulateOneDay();
}

@Override
void addObs() {
    super.addObs();
    abandoned.add (numAbandoned);
    serviceLevel.add (numGoodSL, numServed + numAbandonedAfterAWT);
}

@Override
void simulate (int days) {
    abandoned.init();

```

```

        serviceLevel.init();
        super.simulate (days);
    }

    @Override
    public void printStatistics() {
        System.out.println (abandoned.reportAndCIStudent (LEVEL, 3));
        super.printStatistics();
        System.out.println (serviceLevel.reportAndCIDelta (LEVEL, 3));
    }

    public static void main (String[] args) {
        final MMCAb s = new MMCAb();
        final Chrono timer = new Chrono();
        s.simulate (NUMDAYS);
        System.out.println ("CPU time: " + timer.format());
        s.printStatistics();
    }
}

```

Patience times are i.i.d. exponential variables with mean $1/\nu$, and are generated the same way as service times. To support this aspect, new fields containing the abandonment rate ν as well as a random variate generator for the patience time are defined. Then, a new counter and a new statistical collector are declared for the number of abandoned contacts. A statistical probe is also defined for the new service level estimator. The logic of the main method is the same as for the previous example, except it constructs an instance of `MMCAb` rather than `MMC`.

The constructor first calls the superclass' constructor to create the contact center, and alters the constructed system in the following ways. When contacts are created by the factory, they must now be assigned a patience time. To achieve this result, the factory bound to the arrival process is replaced by a new one called `MyContactFactoryAb`. A second exited-contact listener is also connected to the router in order to count abandoned contacts. The old listener, registered by the superclass, as well as the new one we have just created, will both be called by the router when broadcasting an exited contact. This way, is is not necessary to completely rewrite the event-processing code.

The `simulate` method needs to be overridden to initialize the added collectors. `simulateOneDay` is also overridden to reset the new counter at the beginning of each replication. All the code in the superclass is reused, and overridden methods are called instead of old ones.

`MyContactFactoryAb` extends the previous contact factory to set the patience time of newly-constructed contacts. The `newInstance` method of the superclass is first used to create the contact, then a patience time is set in a way similar to the service time.

Abandonment is managed automatically by the waiting queue as follows. Rather than contact instances, the queue stores objects representing simulation events happening at the

time of automatic removal, e.g., abandonment, disconnection, transfer to another queue, etc. In the previous subsection, these *dequeue events* were never scheduled, because patience times default to infinity if unspecified. The patience time, more generally the *maximal queue time*, is by default extracted from the contacts, the same way as the service time.

Listing 4: Results of the program MMCAb

```

CPU time: 0:0:1.75
REPORT on Tally stat. collector ==> Number of contacts having abandoned
  num. obs.      min      max      average      standard dev.
    1000      20.000    124.000     58.609      15.295
95.0% confidence interval for mean (student): ( 57.660, 59.558 )

REPORT on Tally stat. collector ==> Number of contacts in target
  num. obs.      min      max      average      standard dev.
    1000    1785.000    2028.000    1906.008      35.828
95.0% confidence interval for mean (student): ( 1903.785, 1908.231 )

REPORT on Tally stat. collector ==> Service level
  func. of averages      standard dev.      num. obs.
          0.975          7.9E-3          1000
95.0% confidence interval for function of means: ( 0.974, 0.975 )

REPORT on Tally stat. collector ==> Speed of answer
  func. of averages      standard dev.      num. obs.
          0.024          6.0E-3          1000
95.0% confidence interval for function of means: ( 0.023, 0.024 )

REPORT on Tally stat. collector ==> Occupancy ratio
  func. of averages      standard dev.      num. obs.
          0.675          0.019          1000
95.0% confidence interval for function of means: ( 0.674, 0.677 )

REPORT on Tally stat. collector ==> Service level with abandonment
  func. of averages      standard dev.      num. obs.
          0.968          9.9E-3          1000
95.0% confidence interval for function of means: ( 0.968, 0.969 )

```

The `MyContactMeasuresAb` class defines the `dequeued` method to count abandoned contacts. In a way similar to `served` in `MyContactMeasures`, this method takes an object representing a dequeue event to receive maximal information. An abandoned contact is counted, and if the waiting time is greater than or equal to s , a contact having abandoned after the acceptable waiting time is added. In contrast with the contact factory, `MyContactMeasuresAb` has no inheritance relationship with its counterpart in MMC. Both inner classes

are independent observers performing complementary tasks.

The `addObs` method is overridden to collect the number of abandoned contacts and to compute service level differently. The superclass' method is called and adds observations in the old and inherited collectors.

Finally, `printStats` is overridden to take the number of abandoned contacts into account. Listing 4 presents the results of the extended program. Abandonment increases the service level, because it reduces the size of the waiting queue and allows customers to be served more quickly. The occupancy ratio of agents is reduced, because some customers served in the $M/M/c$ system abandon.

1.3 Clearing the queue at the end of the day

If the contact center was very busy, a big queue could build up during the day. This would result in agents working for a long time after their shifts. To make the model more realistic without supporting abandonment, when the contact center closes, queued contacts may be disconnected instead of being served. Agents still terminate their services, but the queued contacts are not served. To implement this, at the end of the simulation, the `queue.clear` method is called to remove all contacts. This method accepts a *dequeue type* giving the reason why contacts are removed. The dequeue type 0 is reserved for contacts to be served while dequeue type 1 is used for abandoned contacts. In this example, we clear the queue with an indicator different from 1 to permit statistical collectors to distinguish abandonment from disconnection. Listing 5 presents a second extension of the $M/M/c$ queue, with support for disconnection. This new aspect has an impact on the original service level estimator. We therefore define a new collector for a corrected estimator including the number of disconnected contacts in its denominator.

Listing 5: An extension of the $M/M/c$ model supporting disconnections

```
import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;
import umontreal.iro.lecuyer.stat.FunctionOfMultipleMeansTally;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.util.Chrono;
import umontreal.iro.lecuyer.util.RatioFunction;

public class MMCClrQ extends MMC {
    int numDisconnected;
    Tally disconnected = new Tally ("Number of disconnected contacts");
    FunctionOfMultipleMeansTally serviceLevel = new FunctionOfMultipleMeansTally
        (new RatioFunction(), "Service level with disconnected contacts", 2);
```

```
MMCClrQ() {
    super();
    router.addExitedContactListener (new MyContactMeasuresClr());
}

class MyContactMeasuresClr implements ExitedContactListener {
    public void blocked (Router router, Contact contact, int bType) {}
    public void dequeued (Router router, DequeueEvent ev) {
        ++numDisconnected;
    }
    public void served (Router router, EndServiceEvent ev) {}
}

@Override
void endSim() { super.endSim();    queue.clear (5); }

@Override
void simulateOneDay() {
    numDisconnected = 0;
    super.simulateOneDay();
}

@Override
void addObs() {
    super.addObs();
    disconnected.add (numDisconnected);
    serviceLevel.add (numGoodSL, numServed + numDisconnected);
}

@Override
void simulate (int days) {
    disconnected.init();
    serviceLevel.init();
    super.simulate (days);
}

@Override
public void printStatistics() {
    System.out.println (disconnected.reportAndCISTudent (LEVEL, 3));
    super.printStatistics();
    System.out.println (serviceLevel.reportAndCIDelta (LEVEL, 3));
}

public static void main (String[] args) {
    final MMCClrQ s = new MMCClrQ();
    final Chrono timer = new Chrono();
    s.simulate (NUMDAYS);
}
```

```

    System.out.println ("CPU time: " + timer.format());
    s.printStatistics();
}
}

```

Listing 6: Results of the program MMCClrQ

```

CPU time: 0:0:1.58
REPORT on Tally stat. collector ==> Number of disconnected contacts
  num. obs.      min      max      average      standard dev.
    1000      0.000    20.000      0.433      1.688
95.0% confidence interval for mean (student): (    0.328,    0.538 )

REPORT on Tally stat. collector ==> Number of contacts in target
  num. obs.      min      max      average      standard dev.
    1000    1588.000    1964.000    1816.112      57.042
95.0% confidence interval for mean (student): ( 1812.572, 1819.652 )

REPORT on Tally stat. collector ==> Service level
  func. of averages      standard dev.      num. obs.
        0.902        0.033        1000
95.0% confidence interval for function of means: (    0.900,    0.904 )

REPORT on Tally stat. collector ==> Speed of answer
  func. of averages      standard dev.      num. obs.
        0.099        0.045        1000
95.0% confidence interval for function of means: (    0.096,    0.102 )

REPORT on Tally stat. collector ==> Occupancy ratio
  func. of averages      standard dev.      num. obs.
        0.695        0.023        1000
95.0% confidence interval for function of means: (    0.694,    0.697 )

REPORT on Tally stat. collector ==> Service level with disconnected contacts
  func. of averages      standard dev.      num. obs.
        0.902        0.033        1000
95.0% confidence interval for function of means: (    0.900,    0.904 )

```

As it was made for abandonment in the previous example, the base *M/M/c* program is extended to support disconnection. This time, a new contact factory is not needed. As with the previous example, a second observer is connected to the router to count additional events. The exited-contact listener simply counts a disconnection when a contact leaves the queue. However, if abandonment was additionally supported, it would be important to

adapt the exited-contact listener to distinguish abandoned and disconnected contacts. The `endSim` method is overridden to clear the waiting queue.

Listing 6 presents statistical result of this program. Clearing the waiting queue does not have a great impact on the performance of the contact center, because the queue at the end of the day is not too large.

1.4 Limiting the queue capacity

The capacity of a real system is never infinite. For example, call centers own a certain number of phone lines, and a caller arriving at a time when all lines are busy cannot be served or wait in queue; it is *blocked* by the system. To support this aspect, the `Router` class defines the `setTotalQueueCapacity` method to limit the number of queued contacts. Blocked contacts are notified to the `blocked` method of registered exited-contact listeners and can be counted too. With this mechanism, a limitation can be enforced on the capacity over all waiting queues only. In section 4, we will see a more general way of limiting the capacity of a contact center, by using trunk groups.

Listing 7 presents an extension of the $M/M/c$ basic model with limited queue capacity. The program is similar to examples in previous subsections: we extend the `MMC` class to alter the contact center and add new statistical counters. The estimator of the service level is also modified to add the number of blocked contacts in its denominator. This time, the `blocked` method of the exited-contact listener is defined and simply increments the counter for the number of blocked contacts. The `bType` indicator may be useful to get the reason of blocking.

Listing 7: An extension of the $M/M/c$ model with limited queue capacity

```
import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;
import umontreal.iro.lecuyer.stat.FunctionOfMultipleMeansTally;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.util.Chrono;
import umontreal.iro.lecuyer.util.RatioFunction;

public class MMCBlocked extends MMC {
    static final int QUEUECAPACITY = 5;
    int numBlocked;
    Tally blocked = new Tally ("Number of blocked contacts");
    FunctionOfMultipleMeansTally serviceLevel = new FunctionOfMultipleMeansTally
        (new RatioFunction(), "Service level with blocked contacts", 2);

    MMCBlocked() {
```

```

    super();
    router.addExitedContactListener (new MyContactMeasuresBlocked());
    router.setTotalQueueCapacity (QUEUECAPACITY);
}

class MyContactMeasuresBlocked implements ExitedContactListener {
    public void blocked (Router router, Contact contact, int bType) {
        ++numBlocked; }
    public void dequeued (Router router, DequeueEvent ev) {}
    public void served (Router router, EndServiceEvent ev) {}
}

@Override
void simulateOneDay() {
    numBlocked = 0;
    super.simulateOneDay();
}

@Override
void addObs() {
    super.addObs();
    blocked.add (numBlocked);
    serviceLevel.add (numGoodSL, numBlocked + numServed);
}

@Override
void simulate (int days) {
    blocked.init();
    serviceLevel.init();
    super.simulate (days);
}

@Override
public void printStatistics() {
    System.out.println (blocked.reportAndCISudent (LEVEL, 3));
    super.printStatistics();
    System.out.println (serviceLevel.reportAndCIDelta (LEVEL, 3));
}

public static void main (String[] args) {
    final MMCBlocked s = new MMCBlocked();
    final Chrono timer = new Chrono();
    s.simulate (NUMDAYS);
    System.out.println ("CPU time: " + timer.format());
    s.printStatistics();
}
}

```

Listing 8 displays the statistical results of the new program. Limiting the capacity has an effect similar to abandonment by limiting the number of contacts to serve.

Listing 8: Results of the program MCBBlocked

```
CPU time: 0:0:1.55
REPORT on Tally stat. collector ==> Number of blocked contacts
  num. obs.      min      max      average      standard dev.
    1000      0.000    62.000     18.655         9.831
95.0% confidence interval for mean (student): ( 18.045, 19.265 )

REPORT on Tally stat. collector ==> Number of contacts in target
  num. obs.      min      max      average      standard dev.
    1000    1718.000    1978.000    1854.284        42.176
95.0% confidence interval for mean (student): ( 1851.667, 1856.901 )

REPORT on Tally stat. collector ==> Service level
  func. of averages      standard dev.      num. obs.
        0.929          0.019          1000
95.0% confidence interval for function of means: ( 0.928, 0.930 )

REPORT on Tally stat. collector ==> Speed of answer
  func. of averages      standard dev.      num. obs.
        0.060          0.016          1000
95.0% confidence interval for function of means: ( 0.059, 0.061 )

REPORT on Tally stat. collector ==> Occupancy ratio
  func. of averages      standard dev.      num. obs.
        0.689          0.021          1000
95.0% confidence interval for function of means: ( 0.688, 0.690 )

REPORT on Tally stat. collector ==> Service level with blocked contacts
  func. of averages      standard dev.      num. obs.
        0.920          0.022          1000
95.0% confidence interval for function of means: ( 0.919, 0.922 )
```

1.5 Preparing for parallel simulations

As mentioned in the overview, most examples of this guide use the `Sim` static class for simplicity. But if several instances of a simulator are needed, with one instance assigned to one thread, each call to static methods in the `Sim` class must be replaced with a call to methods in an instance of `Simulator`. The program in Listing 9 is an adaptation of Listing 1 showing how to do this.

Listing 9: An extension of the *M/M/c* model using Simulator

```

import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.contact.ContactArrivalProcess;
import umontreal.iro.lecuyer.contactcenters.contact.ContactFactory;
import umontreal.iro.lecuyer.contactcenters.contact.PoissonArrivalProcess;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.queue.StandardWaitingQueue;
import umontreal.iro.lecuyer.contactcenters.queue.WaitingQueue;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.contactcenters.router.QueuePriorityRouter;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.server.AgentGroup;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;
import umontreal.iro.lecuyer.contactcenters.server.GroupVolumeStat;
import umontreal.iro.lecuyer.randvar.ExponentialGen;
import umontreal.iro.lecuyer.rng.MRG32k3a;
import umontreal.iro.lecuyer.simevents.Event;
import umontreal.iro.lecuyer.simevents.Simulator;
import umontreal.iro.lecuyer.simevents.UnusableSimulator;
import umontreal.iro.lecuyer.stat.FunctionOfMultipleMeansTally;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.util.Chrono;
import umontreal.iro.lecuyer.util.RatioFunction;

public class MMCSim {
    // Input data (times are in minutes)
    static final double LAMBDA      = 4.2;      // Arrival rate
    static final double MU          = 0.5;      // Service rate
    static final double AWT         = 20/60.0;  // Acceptable waiting time
    static final int  NUMAGENTS     = 12;
    static final int[] TYPETOGROUPMAP = { { 0 } };
    static final int[] GROUPTOTYPEMAP = { { 0 } };
    static final double LEVEL       = 0.95;     // Level of conf. int.
    static final double DAYLENGTH   = 8.0*60.0; // Eight hours
    static final int  NUMDAYS       = 1000;

    // Contact center components
    Simulator sim;
    ContactArrivalProcess arrivProc;
    AgentGroup agents;
    WaitingQueue queue;
    Router router;

    // Service time generator
    ExponentialGen sgen = new ExponentialGen (new MRG32k3a(), MU);

```

```

// Counters and probe used during replications
int numGoodSL, numServed;
double sumWaitingTimes;
GroupVolumeStat vstat;
double Nb, N;

// Collectors with one obs./rep.
Tally goodSL = new Tally ("Number of contacts in target");
FunctionOfMultipleMeansTally serviceLevel = new FunctionOfMultipleMeansTally
    (new RatioFunction(), "Service level", 2),
    occupancy = new FunctionOfMultipleMeansTally
    (new RatioFunction(), "Occupancy ratio", 2),
    speedOfAnswer = new FunctionOfMultipleMeansTally
    (new RatioFunction(), "Speed of answer", 2);

MMCSim() {
    sim = new Simulator();
    arrivProc = new PoissonArrivalProcess
        (sim, new MyContactFactory(), LAMBDA, new MRG32k3a());
    agents = new AgentGroup (NUMAGENTS);
    queue = new StandardWaitingQueue();
    router = new QueuePriorityRouter (TYPETOGROUPMAP, GROUPTOTYPEMAP);
    arrivProc.addNewContactListener (router);
    router.setAgentGroup (0, agents);
    router.setWaitingQueue (0, queue);
    router.addExitedContactListener (new MyContactMeasures());
    vstat = new GroupVolumeStat (sim, agents);
}

// Creates the new contacts
class MyContactFactory implements ContactFactory {
    public Contact newInstance() {
        final Contact contact = new Contact (sim);
        contact.setDefaultServiceTime (sgen.nextDouble());
        return contact;
    }
}

// Updates counters when a contact exits
class MyContactMeasures implements ExitedContactListener {
    public void blocked (Router router, Contact contact, int bType) {}
    public void dequeued (Router router, DequeueEvent ev) {}
    public void served (Router router, EndServiceEvent ev) {
        ++numServed;
        final double qt = ev.getContact().getTotalQueueTime();
        if (qt < AWT) ++numGoodSL;
        sumWaitingTimes += qt;
    }
}

```



```

    }
}

class EndSimEvent extends Event {
    public EndSimEvent (Simulator sim) { super (sim); }
    @Override
    public void actions() { endSim(); }
}

void endSim() {
    arrivProc.stop();
    Nb = vstat.getStatNumBusyAgents().sum();    // Int. for N_b(t)
    N = vstat.getStatNumAgents().sum();         // Int. for N_0(t)
}

void simulateOneDay() {
    sim.init();    // Initialize clock and clear event list
    new EndSimEvent (sim).schedule (DAYLENGTH);
    arrivProc.init();    agents.init();    queue.init();
    numServed = numGoodSL = 0;
    sumWaitingTimes = 0;
    vstat.init();
    arrivProc.start();
    sim.start();    // Simulation runs here
    addObs();
}

void addObs() {
    goodSL.add (numGoodSL);
    serviceLevel.add (numGoodSL, numServed);
    speedOfAnswer.add (sumWaitingTimes, numServed);
    occupancy.add (Nb, N);
}

void simulate (int days) {
    goodSL.init();
    serviceLevel.init();
    occupancy.init();
    for (int r = 0; r < days; r++) simulateOneDay();
}

public void printStatistics() {
    System.out.println (goodSL.reportAndCIStudent (LEVEL, 3));
    System.out.println (serviceLevel.reportAndCIDelta (LEVEL, 3));
    System.out.println (speedOfAnswer.reportAndCIDelta (LEVEL, 3));
    System.out.println (occupancy.reportAndCIDelta (LEVEL, 3));
}

```

```
public static void main (String[] args) {
    Simulator.defaultSimulator = new UnusableSimulator();
    final MMCSim s = new MMCSim();
    final Chrono timer = new Chrono();
    s.simulate (NUMDAYS);
    System.out.println ("CPU time: " + timer.format());
    s.printStatistics();
}
}
```

First, a field of type `Simulator` named `sim` is added to the program. This simulator encapsulates a simulation clock and event list specific to a `MMCSim` instance rather than having a clock and a list shared among all instances. Calls to `Sim.init` and `Sim.start` in the program are then replaced with calls to `sim.init` and `sim.start`.

The simulator must be passed to most objects of `ContactCenters`. This includes contacts, arrival processes, statistical collectors for agent groups, and any user-defined event. Consequently, we had to add a constructor to the `EndSimEvent` class in order to pass the simulator, and change how `arrivProc` and `vcalc` are constructed. The `MyContactFactory` class is also modified to pass the simulator to each new contact. Agent groups and waiting queues do not need the simulator, because they use the one assigned to the contacts they process.

If one forgets to give the simulator to an object that needs one, no compile or run time errors occur by default. The object in question silently uses the default simulator, which could result in unexecuted simulation events or other unpredictable errors. To help in preventing this, we would like the program to crash as soon as possible if the default simulator is used. This can be done by replacing the default simulator with a dummy unusable object whose each method throws an `UnsupportedOperationException`; we do this at the beginning of the `main` method in the program. If the program does not throw an exception, one can then be relatively confident that the default simulator is never used.

2 A simplified call center with multiple periods

This example comes from the SSJ User's Guide [8] and models a call center with a single call type, a single agent group, but multiple periods. Each day, the center operates for P hours. The arrival rate of calls and the number of agents answering them vary during the day, but they are constant within each hour.

The simulation time T of a replication is divided into $P + 2$ periods. We assume that the contact center is opened during P *main periods*. Main period p , for $p = 1, \dots, P$, corresponds to time interval $[t_{p-1}, t_p)$, where $0 \leq t_0 < \dots < t_P$. Since the simulation may start before the contact center opens and stop after it closes, two extra periods are defined. During the *preliminary period* $[0, t_0)$, the center is closed and no service occurs, but arrivals may start during this period for a queue to build up. During *wrap-up period* $[t_P, T]$, no arrival occurs, but in-progress services are terminated and queued contacts are processed.

Calls arrive following a Poisson process with piecewise-constant randomized arrival rate $B\lambda_p$ during period p , where λ_p is constant and B is a random variable having the gamma distribution with parameters (α_0, α_0) . Thus, B has mean 1 and variance $1/\alpha_0$, and it represents the busyness of the day. The day is more busy than usual when $B > 1$ and less busy than usual when $B < 1$.

Calls that cannot be served immediately are put in a FIFO queue as in the previous example, but callers can abandon. The i.i.d. patience times are generated as follows: with probability ρ , the patience time is 0, i.e., the caller abandons if he cannot be served immediately. With probability $1 - \rho$, the patience time is exponential with mean $1/\nu$. Service times are i.i.d. gamma random variables with shape parameter α , scale parameter β , and mean α/β .

During main period p , N_p agents are available to serve calls. If, at the end of period p , the number of busy agents is larger than N_{p+1} , ongoing calls are completed, but new calls are not accepted until the number of busy agents is smaller than N_{p+1} . During the preliminary period, there is no agent whereas for the wrap-up period, $N_{P+1} = N_P$.

We are interested in estimating the long-term expected waiting time among all calls, the fraction of calls having waited less than s seconds, and the fraction of calls having abandoned. The service level estimated by this program is defined as

$$g_2(s) = \frac{\mathbb{E}[S_G(s) + L_G(s)]}{\mathbb{E}[A]}, \quad (5)$$

which differs from (4). Let W_L be the sum of waiting times for calls having abandoned. If $W = W_S + W_L$ is the total waiting time of all calls and $A = X + Y$ is the number of arrivals, the estimated performance measures are $\mathbb{E}[W]/\mathbb{E}[A]$, $\mathbb{E}[S_G(s) + L_G(s)]/\mathbb{E}[A]$, and $\mathbb{E}[L]/\mathbb{E}[A]$.

The ContactCenters program presented on Listing 10 is simpler than the one presented in SSJ's User's Guide, because many complex aspects are hidden in the library.

Listing 10: A call center with multiple periods

```

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.StringTokenizer;

import umontreal.iro.lecuyer.contactcenters.PeriodChangeEvent;
import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.contact.ContactFactory;
import umontreal.iro.lecuyer.contactcenters.contact.PiecewiseConstantPoissonArrivalProcess;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.queue.StandardWaitingQueue;
import umontreal.iro.lecuyer.contactcenters.queue.WaitingQueue;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.contactcenters.router.QueuePriorityRouter;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.server.AgentGroup;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;

import umontreal.iro.lecuyer.probdist.ExponentialDist;
import umontreal.iro.lecuyer.probdist.GammaDist;
import umontreal.iro.lecuyer.randvar.GammaAcceptanceRejectionGen;
import umontreal.iro.lecuyer.randvar.GammaGen;
import umontreal.iro.lecuyer.rng.MRG32k3a;
import umontreal.iro.lecuyer.rng.RandomStream;
import umontreal.iro.lecuyer.simevents.Sim;
import umontreal.iro.lecuyer.stat.Tally;

public class CallCC {
    static final double HOUR = 3600.0; // Time is in seconds.

    // Data
    // Arrival rates are per hour, service and patience times are in seconds.
    int numDays; // Number of days to simulate.
    double openingTime; // Opening time of the center (in hours).
    int numPeriods; // Number of working periods (hours) in the day.
    int[] numAgents; // Number of agents for each period.
    double[] lambda; // Base arrival rate lambda_j for each j.
    double alpha0; // Parameter of gamma distribution for W.
    double p; // Probability that patience time is 0.
    double nu; // Parameter of exponential for patience time.
    double alpha, beta; // Parameters of gamma service time distribution.
    double s; // Want stats on waiting times smaller than s.

    // Variables
    int nArrivals; // Number of arrivals today;
    int nAbandon; // Number of abandonments during the day.

```

```

int nGoodSL;           // Number of waiting times less than s today.
double nCallsExpected; // Expected number of calls per day.

PeriodChangeEvent pce; // Event marking new periods
PiecewiseConstantPoissonArrivalProcess arrivProc;
WaitingQueue waitingQueue = new StandardWaitingQueue();
AgentGroup agentGroup;
Router router;

RandomStream streamB      = new MRG32k3a(); // For B.
RandomStream streamArr    = new MRG32k3a(); // For arrivals.
RandomStream streamPatience = new MRG32k3a(); // For patience times.
RandomStream streamS      = new MRG32k3a(); // For service times.
GammaGen genServ;        // For service times; created in readData().
GammaGen bgen;           // For busyness; created in readData().

Tally statArrivals = new Tally ("Number of arrivals per day");
Tally statWaits    = new Tally ("Average waiting time per customer");
Tally statWaitsDay = new Tally ("Waiting times within a day");
Tally statGoodSL   = new Tally ("Proportion of waiting times < s");
Tally statAbandon  = new Tally ("Proportion of calls lost");

public CallCC (String fileName) throws IOException {
    readData (fileName);
    pce = new PeriodChangeEvent (HOURL, numPeriods + 2, openingTime*HOURL);
    arrivProc = new PiecewiseConstantPoissonArrivalProcess
        (pce, new MyContactFactory(), lambda, streamArr);
    arrivProc.setNormalizing (true);
    agentGroup = new AgentGroup (pce, numAgents);
    router = new QueuePriorityRouter (new int[][] { { 0 } }, new int[][] { { 0 } });
    arrivProc.addNewContactListener (router);
    router.setWaitingQueue (0, waitingQueue);
    router.setAgentGroup (0, agentGroup);
    router.addExitedContactListener (new MyContactMeasures());
}

// Creates new contacts
class MyContactFactory implements ContactFactory {
    public Contact newInstance() {
        final Contact contact = new Contact();
        contact.setDefaultServiceTime (genServ.nextDouble());
        contact.setDefaultPatienceTime (genPatience());
        return contact;
    }
}

// Updates counters

```

```

class MyContactMeasures implements ExitedContactListener {
    public void blocked (Router router, Contact contact, int bType) {}
    public void dequeued (Router router, DequeueEvent ev) {
        nArrivals++; nAbandon++;
        final double qt = ev.getContact().getTotalQueueTime();
        if (qt < s) nGoodSL++;
        statWaitsDay.add (qt);
    }
    public void served (Router router, EndServiceEvent ev) {
        nArrivals++;
        final double qt = ev.getContact().getTotalQueueTime();
        if (qt < s) nGoodSL++;
        statWaitsDay.add (qt);
    }
}

public double generPatience() {
    if (agentGroup.getNumFreeAgents() > 0) return 0;
    // Generates the patience time for a call.
    final double u = streamPatience.nextDouble();
    if (u <= p) return 0.0;
    else return ExponentialDist.inverseF (nu, (1.0-u) / (1.0-p));
}

void readData (String fileName) throws IOException {
    // Reads data and construct arrays.
    final BufferedReader input = new BufferedReader (new FileReader (fileName));
    StringTokenizer line = new StringTokenizer (input.readLine());
    openingTime = Double.parseDouble (line.nextToken());
    line = new StringTokenizer (input.readLine());
    numPeriods = Integer.parseInt (line.nextToken());

    numAgents = new int[numPeriods+2];
    lambda = new double[numPeriods+2];
    nCallsExpected = 0.0;
    for (int j=0; j < numPeriods; j++) {
        line = new StringTokenizer (input.readLine());
        numAgents[j+1] = Integer.parseInt (line.nextToken());
        lambda[j+1] = Double.parseDouble (line.nextToken());
        nCallsExpected += lambda[j+1];
    }
    numAgents[numAgents.length - 1] = numAgents[numAgents.length - 2];
    line = new StringTokenizer (input.readLine());
    alpha0 = Double.parseDouble (line.nextToken());
    line = new StringTokenizer (input.readLine());
    p = Double.parseDouble (line.nextToken());
    line = new StringTokenizer (input.readLine());

```

```

    nu = Double.parseDouble (line.nextToken());
    line = new StringTokenizer (input.readLine());
    alpha = Double.parseDouble (line.nextToken());
    line = new StringTokenizer (input.readLine());
    beta = Double.parseDouble (line.nextToken());
    // genServ can be created only after its parameters are known.
    genServ = new GammaAcceptanceRejectionGen (streamS, alpha, beta);
                                   // Faster than inversion.
    bgen = new GammaGen (streamB, new GammaDist (alpha0, alpha0, 8));
    line = new StringTokenizer (input.readLine());
    s = Double.parseDouble (line.nextToken());
    input.close();
}

void simulateOneDay() {
    Sim.init();
    statWaitsDay.init();
    nArrivals = nAbandon = nGoodSL = 0;
    pce.init();
    arrivProc.init (bgen.nextDouble());
    agentGroup.init();
    waitingQueue.init();
    arrivProc.start();
    pce.start();
    Sim.start();    // Here the simulation is running...
    pce.stop();
    statArrivals.add (nArrivals);
    statAbandon.add (nAbandon / nCallsExpected);
    statGoodSL.add (nGoodSL / nCallsExpected);
    statWaits.add (statWaitsDay.sum() / nCallsExpected);
}

void simulate (int numDays) {
    statArrivals.init();    statAbandon.init();
    statGoodSL.init();    statWaits.init();
    for (int r=1; r <= numDays; r++) simulateOneDay();
}

void printStatistics() {
    System.out.println ("\n Num. calls expected = " + nCallsExpected + "\n");
    System.out.println (statArrivals.reportAndCISudent (0.9, 3));
    System.out.println (statWaits.reportAndCISudent (0.9, 3));
    System.out.println (statGoodSL.reportAndCISudent (0.9, 3));
    System.out.println (statAbandon.reportAndCISudent (0.9, 4));
}

static public void main (String[] args) throws IOException {

```

```

    final CallCC cc = new CallCC ("CallCenter.dat");
    cc.simulate (1000);
    cc.printStatistics();
}
}

```

To be compatible with the SSJ example, this program reads the parameters from a file with the same format. However, the used file format is sequential, thus inappropriate for complex models: input errors cannot be detected, and the format cannot be extended easily. A properties file, or an hierarchical format such as XML, is strongly recommended instead of this format. Parameters are read by `readData`, which is the exact copy of the method found in the SSJ example, and stored in fields used for constructing the system. In this program, the main time unit is the second rather than the minute.

When reading λ_p and N_p , $\mathbb{E}[A] = \sum_{p=1}^P \lambda_p$ is computed. The arrival rate for the two extra periods is set to 0. After the data is read, the service times generator `genServ` as well as the busyness generator `bgen` are constructed.

The *period-change event* is then created to divide the simulation horizon into $P + 2$ periods. This event is used to notify *period-change listeners* (some arrival processes, agent groups, custom listeners, etc.) when a new period starts. It also provides methods to convert simulation times to period indices and get the start and end times of a specific period. It can manage fixed-sized main periods as well as variable-sized ones. Note that as any other simulation event, the period-change event has an associated instance of the `Simulator` class. If a `Simulator` instance is created as in section 1.5, it must therefore be passed explicitly to the constructor when creating the period-change event.

The Poisson arrival process is constructed with a period-change event, a contact factory, an array of $P + 2$ arrival rates, and a random stream. This process automatically registers as a period-change listener to update the arrival rate at the beginning of periods. The arrival rates entered in the input file provide the number of calls for each hour while the Poisson arrival process needs these rates to be relative to one second. Since the period duration corresponds to one hour, i.e., `HOURL` seconds, input data could also be considered to provide the arrival rate per period. We can then get the appropriate scaling by turning normalization on for the arrival process. When normalization is enabled, the arrival process automatically divides arrival rates by the period duration before using them. If it is turned off (the default), arrival rates are not transformed.

The agent group is constructed with the period-change event and an array of $P + 2$ elements containing the number of agents for each period. As the arrival process, the agent group is automatically registered as a period-change listener for the number of members to be updated at the beginning of periods.

As in the previous section, the router is constructed to associate the single agent group with the single waiting queue. The same connections as for the $M/M/c$ queue are needed between the components of the system.

The `simulate` method initializes the statistical collectors and calls `simulateOneDay` n times to perform the replications. The latter method initializes the simulation clock and all counters. It initializes the period-change event, which resets the current period to 0. The arrival process is initialized with a new value of B , which is obtained using `bgen`. Even though the arrival process is started before calling `Sim.start`, thus at time 0, the first arrival occurs after $t_0 \geq 0$, because the arrival rate is 0 during the preliminary period. When the period-change event is started, using `pce.start`, it is scheduled at time t_0 to open the call center. The `Sim.start` method is then called, which starts executing events.

After the simulation is finished, it is recommended to call the `stop` method of `PeriodChangeEvent` for each period-change listener to be notified about the end of the simulation which is not scheduled as a period-change event. Observations are added to the appropriate tallies before `simulateOneDay` exits.

The contact factory is similar to the one in the previous section, except that the patience time is generated by `generPatience`. The exited-contact listener contains code in the `dequeued` and `served` methods to count arrived and abandoned contacts if necessary. In both cases, the waiting time of the contact is obtained and added to a tally, and a good call is counted if the waiting time is smaller than s .

The contact center closes at the beginning of the wrap-up period, i.e., at time t_P . In the $M/M/c$ queue, we had to manually disable the arrival process, because it was stationary. In this example, because the arrival rate is set to 0 in the wrap-up period, the arrival process is shut off automatically. New arrivals do not occur, but all queued contacts are served before the simulation stops. If N_{P+1} was equal to 0 rather than N_P , agents would terminate their ongoing service, but no new service would start during the wrap-up period. As a result, every contact still in queue during this period would have to abandon.

After all replications are terminated, the `printStatistics` method produces a report similar to Listing 11.

Listing 11: Results of the program CallCC

```

Num. calls expected = 1660.0

REPORT on Tally stat. collector ==> Number of arrivals per day
  num. obs.      min      max      average      standard dev.
    1000      460.000    4206.000    1639.507    513.189
90.0% confidence interval for mean (student): ( 1612.789, 1666.225 )

REPORT on Tally stat. collector ==> Average waiting time per customer
  num. obs.      min      max      average      standard dev.
    1000      0.000    570.757    11.834    34.078
90.0% confidence interval for mean (student): ( 10.060, 13.608 )

REPORT on Tally stat. collector ==> Proportion of waiting times < s
  num. obs.      min      max      average      standard dev.
    1000      0.277    1.155    0.853    0.169
90.0% confidence interval for mean (student): ( 0.844, 0.862 )

REPORT on Tally stat. collector ==> Proportion of calls lost
  num. obs.      min      max      average      standard dev.
    1000      0.0000    0.8440    0.0342    0.0608
90.0% confidence interval for mean (student): ( 0.0311, 0.0374 )

```

3 A simple multi-skill contact center

This section presents an example of a contact center with three contact types, two agent groups, and three two-hours periods. Contacts arrive following a non-homogeneous Poisson process with randomized arrival rate $B\lambda_{k,p}$ for contact type k during period p . As in the previous section, B is a gamma busyness factor with parameters (α_0, α_0) .

When a contact arrives, an agent is selected from a group depending on the contact type. Contacts of type 0 can only be served by agents in group 0 while contacts of type 2 can only be served by agents in group 1. Contacts of type 1 are served by agents in group 0, or in group 1 if no agent is free in group 0. For contacts arriving during period p , service times are i.i.d. exponential random variables with mean $1/\mu_p$. Agents in groups are not differentiated, and the total number of agents can change from periods to periods.

A contact that cannot be served immediately is added to a waiting queue corresponding to its type. After a patience time, if service has not started, the contact abandons. For contacts arriving during period p , patience times are i.i.d. exponential variables with mean $1/\nu_p$.

We are interested in the overall service level defined by (4), and the occupancy ratio of the first agent group defined by

$$o_i = \frac{E \left[\int_0^{t_P} N_{B,i}(t) dt \right]}{E \left[\int_0^{t_P} (N_i(t) + N_{G,i}(t)) dt \right]} \quad (6)$$

where $N_{B,i}(t)$ corresponds to the number of busy agents in group i , $N_i(t)$ corresponds to the total number of agents in group i , and $N_{G,i}(t)$ corresponds to the number of ghost agents in group i , at time t . We also estimate $\mathbb{E}[S_{G,k,p}(s)]$ for $k = 0, \dots, K-1$ and $p = 0, \dots, P-1$, the number of contacts meeting the service level target.

3.1 Implementing the model

Listing 12 presents the code implementing this model.

Listing 12: A simple multi-skill contact center

```
import umontreal.iro.lecuyer.contactcenters.MultiPeriodGen;
import umontreal.iro.lecuyer.contactcenters.PeriodChangeEvent;
import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.contact.ContactFactory;
import umontreal.iro.lecuyer.contactcenters.contact.PiecewiseConstantPoissonArrivalProcess;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.queue.StandardWaitingQueue;
import umontreal.iro.lecuyer.contactcenters.queue.WaitingQueue;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.router.SingleFIFOQueueRouter;
```

```

import umontreal.iro.lecuyer.contactcenters.server.AgentGroup;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;
import umontreal.iro.lecuyer.contactcenters.server.GroupVolumeStat;
import umontreal.iro.lecuyer.probdist.GammaDist;
import umontreal.iro.lecuyer.randvar.GammaGen;
import umontreal.iro.lecuyer.randvar.RandomVariateGen;
import umontreal.iro.lecuyer.rng.MRG32k3a;
import umontreal.iro.lecuyer.simevents.Sim;
import umontreal.iro.lecuyer.stat.FunctionOfMultipleMeansTally;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.stat.matrix.MatrixOfTallies;
import umontreal.iro.lecuyer.util.RatioFunction;

public class SimpleMSK {
    // All times are in minutes
    static final int K = 3; // Number of contact types
    static final int I = 2; // Number of agent groups
    static final int P = 3; // Number of periods
    static final double PERIODDURATION = 120.0; // Two hours
    // LAMBDA[k][p] gives the arrival rate for type k in period p
    static final double[][] LAMBDA =
        { { 0, 4.2, 5.3, 3.2, 0 }, { 0, 5.1, 4.3, 4.8, 0 }, { 0, 6.3, 5.2, 4.8, 0 } };
    // Gamma parameter for busyness
    static final double ALPHAO = 28.7;
    // Service rate for each period
    static final double[] MU = { 0.5, 0.5, 0.6, 0.4, 0.4 };
    // Abandonment rate for each period
    static final double[] NU = { 0.3, 0.3, 0.4, 0.2, 0.2 };
    // Acceptable waiting time (20 sec.)
    static final double AWT = 20/60.0;
    // NUMAGENTS[i][p] gives the number of agents for group i in period p
    static final int[][] NUMAGENTS = { { 0, 12, 18, 9, 9 }, { 0, 15, 20, 11, 11 } };
    // Routing table, TYPETOGROUPMAP[k] and GROUPTOTYPEMAP[i] contain ordered lists
    static final int[][] TYPETOGROUPMAP = { { 0 }, { 0, 1 }, { 1 } };
    static final int[][] GROUPTOTYPEMAP = { { 1, 0 }, { 2, 1 } };
    static final double LEVEL = 0.95; // Level for confidence intervals
    static final int NUMDAYS = 1000; // Number of replications

    PeriodChangeEvent pce; // Event marking the beginning of each period
    PiecewiseConstantPoissonArrivalProcess[] arrivProc
        = new PiecewiseConstantPoissonArrivalProcess[K];
    AgentGroup[] groups = new AgentGroup[I];
    WaitingQueue[] queues = new WaitingQueue[K];
    Router router;
    RandomVariateGen sgen; // Service times generator
    RandomVariateGen pgen; // Patience times generator
    RandomVariateGen bgen; // Busyness generator

```

```

// Counters
int numGoodSL, numServed, numAbandoned, numAbandonedAfterAWT;
double[][] numGoodSLKP = new double[K][P];
GroupVolumeStat vstat; // Integral of the occupancy ratio

// statistical collectors
Tally served = new Tally ("Number of served contacts");
Tally abandoned = new Tally ("Number of contacts having abandoned");
MatrixOfTallies<Tally> goodSLKP = MatrixOfTallies.createWithTally (K, P);
FunctionOfMultipleMeansTally serviceLevel = new FunctionOfMultipleMeansTally
    (new RatioFunction(), "Service level", 2);
FunctionOfMultipleMeansTally occupancy = new FunctionOfMultipleMeansTally
    (new RatioFunction(), "Occupancy ratio", 2);

SimpleMSK() {
    goodSLKP.setName ("Number of contacts meeting target service level");
    for (int k = 0; k < K; k++)
        for (int p = 0; p < P; p++) {
            goodSLKP.get (k, p).setName ("Type " + k + ", period " + p);
            goodSLKP.get (k, p).setConfidenceIntervalStudent();
            goodSLKP.get (k, p).setConfidenceLevel (LEVEL);
        }
    // One dummy preliminary period, P main periods, and one wrap-up period,
    // main periods start at time 0.
    pce = new PeriodChangeEvent (PERIODDURATION, P + 2, 0);
    for (int k = 0; k < K; k++) // For each contact type
        arrivProc[k] = new PiecewiseConstantPoissonArrivalProcess
            (pce, new MyContactFactory (k), LAMBDA[k], new MRG32k3a());
    bgen = new GammaGen (new MRG32k3a(), new GammaDist (ALPHA0, ALPHA0));
    for (int i = 0; i < I; i++) groups[i] = new AgentGroup (pce, NUMAGENTS[i]);
    for (int q = 0; q < K; q++) queues[q] = new StandardWaitingQueue();
    sgen = MultiPeriodGen.createExponential (pce, new MRG32k3a(), MU);
    pgen = MultiPeriodGen.createExponential (pce, new MRG32k3a(), NU);
    router = createRouter();
    for (int k = 0; k < K; k++) arrivProc[k].addNewContactListener (router);
    for (int i = 0; i < I; i++) router.setAgentGroup (i, groups[i]);
    for (int q = 0; q < K; q++) router.setWaitingQueue (q, queues[q]);
    router.addExitedContactListener (new MyContactMeasures());
    vstat = new GroupVolumeStat (groups[0]);
}

Router createRouter() {
    return new SingleFIFOQueueRouter (TYPETOGROUPMAP, GROUPTOTYPEMAP);
}

// Creates the new contacts

```

```

class MyContactFactory implements ContactFactory {
    int type;
    MyContactFactory (int type) { this.type = type; }
    public Contact newInstance() {
        final Contact contact = new Contact (type);
        contact.setDefaultServiceTime (sgen.nextDouble());
        contact.setDefaultPatienceTime (pgen.nextDouble());
        return contact;
    }
}

// Updates counters when a contact exits
class MyContactMeasures implements ExitedContactListener {
    public void blocked (Router router, Contact contact, int bType) {}
    public void dequeued (Router router, DequeueEvent ev) {
        ++numAbandoned;
        if (ev.getContact().getTotalQueueTime() >= AWT) ++numAbandonedAfterAWT;
    }
    public void served (Router router, EndServiceEvent ev) {
        ++numServed;
        final Contact contact = ev.getContact();
        if (contact.getTotalQueueTime() < AWT) {
            ++numGoodSL;
            final int period = pce.getPeriod (contact.getArrivalTime()) - 1;
            if (period >= 0 || period < P) ++numGoodSLKP[contact.getTypeId()][period];
        }
    }
}

void simulateOneDay() {
    Sim.init();    pce.init();
    final double b = bgen.nextDouble();
    for (int k = 0; k < K; k++) arrivProc[k].init (b);
    for (int i = 0; i < I; i++) groups[i].init();
    for (int q = 0; q < K; q++) queues[q].init();
    numGoodSL = numServed = numAbandoned = numAbandonedAfterAWT = 0;    vstat.init();
    for (int k = 0; k < K; k++) for (int p = 0; p < P; p++) numGoodSLKP[k][p] = 0;
    for (int k = 0; k < K; k++) arrivProc[k].start();
    pce.start();    Sim.start();    // Simulation runs here
    pce.stop();
    served.add (numServed);    abandoned.add (numAbandoned);    goodSLKP.add (numGoodSLKP);
    serviceLevel.add (numGoodSL, numServed + numAbandonedAfterAWT);
    final double Nb = vstat.getStatNumBusyAgents().sum();    // Integral of N_b0(t)
    final double N = vstat.getStatNumAgents().sum();    // Integral of N_0(t)
    final double Ng = vstat.getStatNumGhostAgents().sum();    // Integral of N_g0(t)
    occupancy.add (Nb, N + Ng);
}

```

```

void simulate (int n) {
    served.init();           abandoned.init();    goodSLKP.init();
    serviceLevel.init();     occupancy.init();
    for (int r = 0; r < n; r++) simulateOneDay();
}

public void printStatistics() {
    System.out.println (served.reportAndCISTudent (LEVEL, 3));
    System.out.println (abandoned.reportAndCISTudent (LEVEL, 3));
    System.out.println (serviceLevel.reportAndCIDelta (LEVEL, 3));
    System.out.println (occupancy.reportAndCIDelta (LEVEL, 3));
    for (int k = 0; k < K; k++)
        System.out.println (goodSLKP.rowReport (k));
}

public static void main (String[] args) {
    final SimpleMSK s = new SimpleMSK();  s.simulate (NUMDAYS);  s.printStatistics();
}
}

```

As with previous examples, a class representing the simulator is defined. The `main` method constructs the simulator, runs a simulation, and prints some statistics.

Fields are declared for contact center's components, statistical counters, etc. For the number of served and abandoned contacts, simple integers are sufficient, but a matrix is needed to get the number of contacts meeting service level target, for each contact type and main period.

As in previous examples, a constructor is responsible for creating all arrival processes, agent groups, waiting queues, etc. The period-change event has a preliminary period of length 0, two-hours main periods, and a wrap-up period with random length. For each contact type, a factory and an arrival process are constructed. In contrast with the previous examples, the factory requires the contact type identifier to be given as an argument. In this example, all the arrival processes use the same period-change event to be notified when a new period starts. Some arrival processes could of course use different period-change events. Constructing the agent groups only requires the period-change event for $N_i(t)$ to be automatically updated, and an array containing the number of agents for each period.

Service and patience times are generated using random variate generators adapted for multiple periods. Such generators use a period-change event to determine the current period and select a period-specific generator to get random values. The generic way to construct them is to create a random variate generator for each period and give the resulting array of generators, with a period-change event, to the constructor of `MultiPeriodGen`. For some distributions, including exponential, helper methods such as `createExponential` are available to construct the generators more conveniently.

Because each waiting queue and agent group is functionally identical, it is the task of the router to decide which contacts to send to which agents or queues, and from which queues contacts must be removed. For the router to be constructed, a type-to-group map and a group-to-type map are needed. The selected `SingleFIFOQueueRouter` class affects how these structures are used. Here, because we model a multi-skill contact center, the data structures for the routing policy become important and can greatly affect the performance of the contact center. The choice of the subclass of `Router` affects how contacts interact with the center.

The `simulateOneDay` method must initialize several arrays of elements, including arrival processes, agent groups, and counters. The same value of B is used for every arrival process, because the busyness is not specific to a contact type. As in the previous example, the period-change event and the arrival processes are started before the simulation starts.

One factory object has been constructed for each arrival process, the only difference between these objects being the value of their `type` field. This avoids the necessity of one factory class for each contact type, which greatly improves scalability. The factory constructs a contact of the appropriate type and generates a service and a patience times.

When a contact of type 0 arrives, the router takes the element 0 of the type-to-group map, which corresponds to an ordered list containing the agent group 0 only. Let $N_{F,i}(t)$ be the number of idle agents in group i available to serve contacts. If $N_{F,0}(t) > 0$, the contact is served immediately. Otherwise, it is added to waiting queue 0. Contacts of type 2 are treated similarly. For contacts of type 1, the router obtains an ordered list containing 0 and 1. If $N_{F,0}(t) > 0$, the contact is served immediately by an agent in group 0. Otherwise, it overflows to the next agent group in the list: if $N_{F,1}(t) > 0$, the contact is served by an agent in group 1. Otherwise, it is added at the end of the queue 1.

When an agent within group 0 becomes free or is added, the router uses the group-to-type map to obtain the ordered list $\{1, 0\}$. The chosen router selects the queued contact with the longest waiting time rather than using the order induced by the list. If the waiting queues accessible for agents in group 0 contain no contact, the agent remains free until new arrivals occur. Similar routing happens for agents in group 1. This is equivalent to managing a single FIFO queue by merging all per-type waiting queues, sorting contacts by increasing arrival times, and removing the first contact the free agent can serve.

If `QueuePriorityRouter` was used as in previous examples, the algorithm for agent selection would be the same, but waiting queues would be scanned sequentially rather than considered as a single FIFO queue. For example, if an agent in group 0 became free, the queue priority router would check for contacts in queue 0 only when queue 1 is empty.

Each contact exiting the system is notified to the registered exited-contact listener. The `blocked` method does nothing because the capacity of the contact center is infinite; no contact is blocked. When a contact leaves the queue without service, it is counted as having abandoned. If its waiting time is greater than or equal to s , a contact having abandoned after the acceptable waiting time is also counted. When a contact is served, a new service is counted. If its waiting time is small enough, it is also counted as a contact meeting service level target.

For a contact to be counted in `numGoodSLKP`, the main period of its arrival must be determined. Arrivals occur in periods $1, \dots, P$, corresponding to main periods $0, \dots, P - 1$. If the main period index, i.e., the period index minus one, is negative or greater than or equal to P , the arrival occurred during the preliminary or wrap-up periods, and the event is ignored. When the arrival occurs during a main period, the appropriate element of the matrix is incremented.

After the simulation stops, the `stop` method of `PeriodChangeEvent` is called, computed observations are added to collectors. To keep the program simple, we estimate the occupancy ratio from times 0 to T rather than t_P . Getting the correct ratio would require the creation of a custom period-change listener to get $\int_0^{t_P} N_{B,0}(t) dt$ and $\int_0^{t_P} (N_0(t) + N_{G,0}(t)) dt$ at time t_P , i.e., at the beginning of wrap-up period. This will be done in a further example, in sections 4.3 and 8. Listing 13 displays the results of the program when performing 1000 independent runs.

3.2 Adding a contact-by-contact trace

For debugging or advanced statistical processing using tools such as SAS or R, it may be needed to get a contact-by-contact trace of the simulation. This can be done easily by using observers, as shown in Listing 14. This example is a simplified version of the trace facility available in the generic simulator of call centers supporting blend and multi-skill systems. It shows how it is possible to extract information from a contact object, and other parts of the simulator.

Listing 14: Contact-by-contact trace added to the simple multi-skill contact center example

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.Writer;

import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;

public class SimpleMSKWithTrace extends SimpleMSK {
    static final int NUMDAYS = 5; // Number of replications
    private TraceManager trace;
    private File traceFile;
    private int currentRep = 0;

    public SimpleMSKWithTrace (File traceFile, int timePrecision) {
```

```

    super ();
    this.traceFile = traceFile;
    trace = new TraceManager (timePrecision);
    router.addExitedContactListener (trace);
}

final class TraceManager implements ExitedContactListener {
    private PrintWriter output;
    private boolean closed = true;
    private int timePrecision;

    public TraceManager (int timePrecision) {
        this.timePrecision = timePrecision;
    }

    public void init (Writer output) {
        if (output == null)
            throw new NullPointerException ();
        this.output = new PrintWriter (new BufferedWriter (output));
        closed = false;
    }

    public void close () {
        output.close ();
        closed = true;
    }

    public void writeHeader () {
        if (closed)
            return;
        final String timeP = "%" + (7 + timePrecision) + "s";
        output.format (" Step      Type  Period " + timeP + " "
            + timeP + "      Outcome  Group " + timeP, "ArvTime",
                "QueueTime", "SrvTime");
        output.println ();
    }

    public void blocked (Router router, Contact contact, int bType) {}

    public void dequeued (Router router, DequeueEvent ev) {
        if (closed)
            return;
        final Contact contact = ev.getContact ();
        final int itr = getStep (contact);
        final int k = contact.getTypeId ();
        final int p = getPeriod (contact);
        final double a = contact.getArrivalTime ();

```

```

        final double q = contact.getTotalQueueTime ();
        final String timeP = "%" + (7 + timePrecision) + "." + timePrecision + "f";
        output.printf ("%6d %6d %6d " + timeP + " " + timeP
            + " %9s %6d " + timeP, itr, k, p, a, q, "Abandoned", -1,
                Double.NaN);
        output.println ();
    }

    public void served (Router router, EndServiceEvent ev) {
        if (closed)
            return;
        final Contact contact = ev.getContact ();
        final int itr = getStep (contact);
        final int k = contact.getTypeId ();
        final int p = getPeriod (contact);
        final double a = contact.getArrivalTime ();
        final double q = contact.getTotalQueueTime ();
        final int i = contact.getLastAgentGroup ().getId ();
        final double s = contact.getTotalServiceTime ();
        final String timeP = "%" + (7 + timePrecision) + "." + timePrecision + "f";
        output.printf ("%6d %6d %6d " + timeP + " " + timeP
            + " %9s %6d " + timeP, itr, k, p, a, q, "Served", i, s);
        output.println ();
    }

    private int getStep (Contact contact) {
        return currentRep;
    }

    public int getPeriod (Contact contact) {
        return pce.getPeriod (contact.getArrivalTime ());
    }
}

@Override
void simulateOneDay () {
    super.simulateOneDay ();
    ++currentRep;
}

@Override
void simulate (int n) {
    try {
        final Writer output = new FileWriter (traceFile);
        trace.init (output);
        trace.writeHeader ();
    }
}

```

```

        catch (final IOException ioe) {
            ioe.printStackTrace ();
        }
        currentRep = 0;
        try {
            super.simulate (n);
        }
        finally {
            trace.close ();
        }
    }

    public static void main (String[] args) {
        final SimpleMSKWithTrace s = new SimpleMSKWithTrace (new File (
            "contactTrace.log"), 3);
        s.simulate (NUMDAYS);
        s.printStatistics ();
    }
}

```

This program behaves the same as the program in the preceding subsection, but it creates a text file named `contactTrace.log` with one line for each processed contact, whether it has abandoned or was served. One can then open the resulting (large) trace file with any text editor, or make a program to parse it. Listing 15 displays ten lines of the trace produced by the program. We cannot display the full trace, because its size is 2MB, even though we have simulated only five days.

The trace contains the following fields:

Step The index of the replication the contact arrived in.

Type The type of the contact.

Period The period the contact arrived in.

ArrvTime The simulation time of the contact's arrival.

QueueTime The waiting time in queue of the contact.

Outcome The outcome of the contact, can be **Served** or **Abandoned**.

Group The group index of the agent who has served the contact, or `-1` for abandoned contacts.

SrvTime The service time of the contact, or `NaN` for unserved contacts.

The program defines a class named `SimpleMSKWithTrace` which extends `SimpleMSK` to inherit the simulation logic of the example in the previous subsection, but it adds a new exited-contact listener to the router for logging contacts. This trace manager, which is initialized at the beginning of the simulation, writes a trace entry to a file for any exiting contact. It does not have any effect on how the contacts are managed by the simulator.

The new subclass defines three fields: the trace manager, the output trace file, and the current replication number. The first two fields are initialized by the constructor, which also attach the trace manager to the router, defined in the superclass `SimpleMSK`. This way, every contact processed by the router is broadcast to the contact trace manager. Moreover, one could disable the contact-by-contact trace simply by unregistering the trace manager from the router.

The last field is updated after each replication by an overridden `simulateOneDay` method, and is used when formatting trace entries.

The overridden `simulate` method opens the trace file at the beginning of the simulation, calls the `simulate` method from the superclass to perform the simulation, and closes the trace at the end. We put the call to `close` into a `finally` block that will be called even if an unexpected exception is thrown by the simulator. This prevents any loss of information caused by an unflushed buffer.

The main part of the program is the `TraceManager` inner class which is an exited-contact listener producing the data written into the trace. A trace manager has an associated print writer used to format trace entries into the output file.

Log lines are constructed by the `dequeued` and `served` methods of `TraceManager`, using the `format` method of `PrintWriter`. Note how the arrival time, the service time, the waiting time, the contact type, etc. can all be extracted from the `Contact` object. The `format` method fills a user-provided pattern by using the values given by the remaining arguments. The print stream takes the system's current locale and the line separator into account.

Arrival times and time durations are formatted with a fixed number `timePrecision` of decimal digits of precision, which is set to 3 in this program. This allows for a better visual formatting of the data in the plain text file. One could easily use another format, e.g., XML, or use JDBC to write data into a database instead of a file.

3.3 Rerouting queued contacts

Agents may sometimes be allowed to serve some types of contacts only after the contacts have waited for some time in queue. For example, we can modify the model of this example to allow agents in group 1 to serve contacts of type 1 only after these contacts have waited more than 12s. This can be done by customizing the router as in Listing 16 for *contact rerouting*, i.e., the router can, after a delay elapses, reprocess a queued contact for a new agent selection with different criteria. Note that the router in `ContactCenters` also supports *agent rerouting* which is not covered in this example but works in a way similar to contact rerouting: if an agent stays free after the end of a service for a certain delay, the router can reprocess the agent to do a new contact selection, with different criteria. Of course, agent rerouting requires the agent groups to keep track of individual agents.

Listing 16: Contact rerouting added to the simple multi-skill contact center example

```

import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.queue.WaitingQueue;
import umontreal.iro.lecuyer.contactcenters.router.AgentGroupSelectors;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.router.SingleFIFOQueueRouter;
import umontreal.iro.lecuyer.contactcenters.server.Agent;
import umontreal.iro.lecuyer.contactcenters.server.AgentGroup;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;

import umontreal.iro.lecuyer.simevents.Sim;

public class SimpleMSKWithRerouting extends SimpleMSK {
    static final int[][] TYPETOGROUPMAP1 = { { 0 }, { 0 }, { 1 } };
    static final int[][] TYPETOGROUPMAP2 = { { 0 }, { 0, 1 }, { 1 } };
    static final double DELAY = 0.2;

    @Override
    Router createRouter () {
        return new MyRouter (TYPETOGROUPMAP1, GROUPTOTYPEMAP);
    }

    class MyRouter extends SingleFIFOQueueRouter {
        public MyRouter (int[][] typeToGroupMap, int[][] groupToTypeMap) {
            super (typeToGroupMap, groupToTypeMap);
        }

        @Override
        protected double getReroutingDelay (DequeueEvent dqEv, int numReroutingsDone) {
            return numReroutingsDone == -1 ? DELAY : -1;
        }

        @Override
        protected EndServiceEvent selectAgent (DequeueEvent dqEv, int numReroutingsDone) {
            final Contact contact = dqEv.getContact ();
            final AgentGroup g = AgentGroupSelectors.selectFirst
                (this, TYPETOGROUPMAP2[contact.getTypeId()]);
            if (g == null)
                return null;
            final EndServiceEvent es = g.serve (contact);
            assert es != null : "AgentGroup.serve should not return null";
            return es;
        }

        @Override
        protected DequeueEvent selectContact (AgentGroup group, Agent agent) {

```

```

    WaitingQueue bestQueue = null;
    final double enqueueTime = Double.POSITIVE_INFINITY;
    for (final int k : groupToTypeMap[group.getId()]) {
        final WaitingQueue queue = getWaitingQueue (k);
        if (queue.isEmpty ())
            continue;
        final DequeueEvent firstContact = queue.getFirst ();
        final double time = firstContact.getEnqueueTime ();
        if (group.getId () == 1 && k == 1 && (Sim.time() - time) < DELAY)
            continue;
        if (time < enqueueTime)
            bestQueue = queue;
    }
    if (bestQueue == null)
        return null;
    else
        return bestQueue.removeFirst (DEQUEUEUETYPE_BEGINSERVICE);
}
}

public static void main (String[] args) {
    final SimpleMSKWithRerouting s = new SimpleMSKWithRerouting();
    s.simulate (NUMDAYS);
    s.printStatistics();
}
}

```

The customized router, named `MyRouter`, extends the `SingleFIFOQueueRouter` used by the original example. As a result, the agent and contact selections are performed the same way as in the original example, but we use a different type-to-group map which allows incoming contacts of type 1 to be served by agents in group 0 only.

Contact rerouting works as follows. When a new contact is notified to the router, an agent is selected by the `selectAgent` methods which is given the new contact as its argument. This method must construct and return an end-service event representing the served contact, or `null` if the contact cannot be served. When the contact cannot be served, it is added into a waiting queue by the `selectWaitingQueue` method of the router. If the contact can be queued, the router obtains its initial *rerouting delay*, i.e., the time after which the queued contact is reprocessed if still in queue. If this delay is negative, no rerouting happens, which is the default. When a positive rerouting delay elapses for a queued contact, the router passes this queued contact to a second `selectAgent` method which accepts the dequeue event along with the number of reroutings done. For the first rerouting, this corresponds to 0. This new method returns an end-service event or `null`, exactly as the ordinary `selectAgent` method. Of course, the scheme for agent selection implemented by this method should differ from the scheme implemented by the ordinary `selectAgent` method. If the contact still cannot be

served, it remains in queue, and a new rerouting delay is required. The process continues until a negative delay is obtained, the contact is served, or abandons.

The initial rerouting delay is obtained using `getReroutingDelay` with `-1` as the number of preceding reroutings. By default, this method always returns `-1`, which results in no rerouting. In our case, we return `DELAY` if the number of reroutings done is `-1`, or `-1` otherwise. The `selectAgent` method for rerouting applies the overflow routing of the original example, which allows contacts of type 1 to be routed to agents in group 1 when no agent is available in group 0.

We also need to override contact selection in order to prevent an agent in group 1 to dequeue a contact of type 1 if its waiting time is smaller than `DELAY`. We therefore override the `selectContact` method of `Router`, which accepts an agent group containing the free agents, and returns the dequeue event representing the contact to serve. Our method scans the waiting queues accessible to the free agent, and records the waiting queue whose first contact has the smallest enqueue time. However, if the free agent is in group 1, the method filters out queue 1 if the first contact has a waiting time smaller than `DELAY`. After the queue is selected, the first contact is removed, and returned. The contact is removed with dequeue type `DEQUEUEUETYPE_BEGINSERVICE` which is used to represent the beginning of the service for a queued contact.

Listing 17 shows that the modified program produces slightly different results: the number of contacts having abandoned is higher and the service level is smaller than with the original model. The simulator behaves as we would expect: some contacts of type 0 have to wait 12s longer in queue before they can be served, and some of them abandon while they were served in the original setting.

Listing 13: Results of the program SimpleMSK

```

REPORT on Tally stat. collector ==> Number of served contacts
  num. obs.      min      max      average      standard dev.
    1000      2448.000    5291.000    4297.305      437.605
95.0% confidence interval for mean (student): ( 4270.150, 4324.460 )

REPORT on Tally stat. collector ==> Number of contacts having abandoned
  num. obs.      min      max      average      standard dev.
    1000         9.000    3805.000     910.335      551.164
95.0% confidence interval for mean (student): ( 876.133, 944.537 )

REPORT on Tally stat. collector ==> Service level
  func. of averages      standard dev.      num. obs.
        0.501          0.128          1000
95.0% confidence interval for function of means: ( 0.494, 0.509 )

REPORT on Tally stat. collector ==> Occupancy ratio
  func. of averages      standard dev.      num. obs.
        0.881          0.058          1000
95.0% confidence interval for function of means: ( 0.877, 0.884 )

Report for Number of contacts meeting target service level
      num obs.      min      max      average      std. dev.      conf. int.
Type 0, period 0  1000      5.000    334.000     170.154      93.722    95.0% ( \
164.338, 175.970)
Type 0, period 1  1000     274.000    782.000     599.334      83.392    95.0% ( \
594.159, 604.509)
Type 0, period 2  1000      0.000    180.000      15.490      27.520    95.0% ( \
13.782, 17.198)

Report for Number of contacts meeting target service level
      num obs.      min      max      average      std. dev.      conf. int.
Type 1, period 0  1000      8.000    484.000     262.936     134.377    95.0% ( \
254.597, 271.275)
Type 1, period 1  1000     255.000    720.000     511.648      86.571    95.0% ( \
506.276, 517.020)
Type 1, period 2  1000      0.000    298.000      34.697      55.230    95.0% ( \
31.270, 38.124)

Report for Number of contacts meeting target service level
      num obs.      min      max      average      std. dev.      conf. int.
Type 2, period 0  1000     15.000    540.000     259.806     155.150    95.0% ( \
250.178, 269.434)
Type 2, period 1  1000     265.000    850.000     613.732      99.046    95.0% ( \
607.586, 619.878)
Type 2, period 2  1000      0.000    282.000      24.888      44.327    95.0% ( \
22.137, 27.639)

```

Listing 15: Sample trace produced by the program SimpleMSKWithTrace

Step	Type	Period	ArvTime	QueueTime	Outcome	Group	SrvTime
0	1	1	0.365	0.000	Served	0	0.012
0	0	1	0.277	0.000	Served	0	0.212
0	0	1	0.162	0.000	Served	0	0.790
0	2	1	0.969	0.000	Served	1	0.567
0	0	1	1.135	0.000	Served	0	1.053
0	0	1	1.339	0.000	Served	0	1.078
0	0	1	1.521	0.000	Served	0	0.941
1	2	1	24.243	0.965	Served	1	0.633
1	1	1	22.958	1.065	Served	0	1.954
1	0	1	24.668	1.104	Served	0	0.215
1	2	1	21.892	0.887	Served	1	3.259
1	1	1	25.111	0.951	Abandoned	-1	NaN
1	0	1	23.158	0.942	Served	0	2.015
1	0	1	21.637	0.802	Served	0	3.714
1	1	1	25.874	0.422	Abandoned	-1	NaN

Listing 17: Results of SimpleMSKWithRerouting

```

REPORT on Tally stat. collector ==> Number of served contacts
  num. obs.      min      max      average      standard dev.
    1000      2430.000    5367.000    4274.426      433.840
95.0% confidence interval for mean (student): ( 4247.504, 4301.348 )

REPORT on Tally stat. collector ==> Number of contacts having abandoned
  num. obs.      min      max      average      standard dev.
    1000      14.000    3720.000     933.214      554.096
95.0% confidence interval for mean (student): ( 898.830, 967.598 )

REPORT on Tally stat. collector ==> Service level
  func. of averages      standard dev.      num. obs.
        0.579          0.113          1000
95.0% confidence interval for function of means: ( 0.572, 0.587 )

REPORT on Tally stat. collector ==> Occupancy ratio
  func. of averages      standard dev.      num. obs.
        0.884          0.057          1000
95.0% confidence interval for function of means: ( 0.880, 0.887 )

Report for Number of contacts meeting target service level
      num obs.      min      max      average      std. dev.      conf. int\
Type 0, period 0  1000      118.000    391.000    298.745      30.117    95.0% ( \
296.876, 300.614)
Type 0, period 1  1000      274.000    791.000    595.270      82.484    95.0% ( \
590.151, 600.389)
Type 0, period 2  1000         4.000    181.000    106.823      29.504    95.0% ( \
104.992, 108.654)

Report for Number of contacts meeting target service level
      num obs.      min      max      average      std. dev.      conf. int\
Type 1, period 0  1000      150.000    487.000    383.203      50.558    95.0% ( \
380.066, 386.340)
Type 1, period 1  1000      255.000    706.000    504.090      82.433    95.0% ( \
498.975, 509.205)
Type 1, period 2  1000         0.000    305.000    121.441      61.286    95.0% ( \
117.638, 125.244)

Report for Number of contacts meeting target service level
      num obs.      min      max      average      std. dev.      conf. int\
Type 2, period 0  1000         9.000    547.000    239.008      158.435    95.0% ( \
229.176, 248.840)
Type 2, period 1  1000      265.000    857.000    611.550      97.645    95.0% ( \
605.491, 617.609)
Type 2, period 2  1000         0.000    282.000     22.298      42.372    95.0% ( \
19.669, 24.927)

```

4 Telethon call center

This example, adapted from Rockwell's Arena Contact Center Edition 8.0, deals with the organization of a one-week pledge drive local public radio station. In this model, on each weekday, from 6AM to 10AM, the 24 phone lines of the contact center accept contacts from donors while donations are managed by 12 volunteers. If a contact cannot be served immediately when it enters the system, it is added to a FIFO queue. The contact leaves the queue if

1. its waiting time is greater than its exponential patience time with mean 2 minutes (abandonment),
2. its waiting time reaches 2 minutes (the contact leaves a message and is disconnected),
3. the contact center closes (disconnection),
4. or a volunteer becomes idle and serves it.

Service times are i.i.d. exponential variables with mean 10 minutes, and arrivals follow a Poisson process whose rate is 50 contacts per hour.

Since the expected number of donors is limited, the contact center must minimize busy signals and abandonment. Key performance measures are blocked and abandoned contacts as well as the expected speed of answer, which is the expected time a contact must wait in queue before it is served. The program also estimates the service level defined as

$$g_3(s) = \frac{\mathbb{E}[S_G(s)]}{\mathbb{E}[A]}, \quad (7)$$

where A is the number of arrivals. It also estimates the occupancy ratio of agents defined by (3).

4.1 Implementing the model

Listing 18: A simulation program implementing the Arena Telethon example

```
import umontreal.iro.lecuyer.contactcenters.ConstantValueGenerator;
import umontreal.iro.lecuyer.contactcenters.PeriodChangeEvent;
import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.contact.ContactFactory;
import umontreal.iro.lecuyer.contactcenters.contact.PiecewiseConstantPoissonArrivalProcess;
import umontreal.iro.lecuyer.contactcenters.contact.TrunkGroup;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.queue.StandardWaitingQueue;
import umontreal.iro.lecuyer.contactcenters.queue.WaitingQueue;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
```

```

import umontreal.iro.lecuyer.contactcenters.router.QueuePriorityRouter;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.server.AgentGroup;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;
import umontreal.iro.lecuyer.contactcenters.server.GroupVolumeStat;
import umontreal.iro.lecuyer.randvar.ExponentialGen;
import umontreal.iro.lecuyer.rng.MRG32k3a;
import umontreal.iro.lecuyer.simevents.Sim;
import umontreal.iro.lecuyer.stat.FunctionOfMultipleMeansTally;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.util.Chrono;
import umontreal.iro.lecuyer.util.RatioFunction;

public class Telethon {
    // Input data (time is in minutes)
    static final double SERVICETIME      = 10;
    static final double PATIENCETIME     = 2;
    static final double AWT               = 1.0;
    static final int P                    = 24*7;
    static final double PERIODDURATION   = 60.0;
    static final int STARTHOUR           = 6;
    static final int ENDFEEDBACK         = 10;
    static final int NUMTRUNKS           = 24; // Number of phone lines
    static final int NUMAGENTS           = 12;
    static final double ARRIVALRATE       = 50;
    static final int[][] TYPETOGROUPMAP   = { { 0 } };
    static final int[][] GROUPTOTYPEMAP   = { { 0 } };
    static final double LEVEL            = 0.95; // Level of confidence intervals
    static final int NUMDAYS              = 1000;

    // Known expectations
    // (ENDFEEDBACK - STARTHOUR) hours per day,
    // Five working days per week
    // ARRIVALRATE contacts per hour
    static final double EXPARRIVALS = ARRIVALRATE*(ENDFEEDBACK - STARTHOUR)*5;
    // Expected value of the integral for the number of agents N(t)
    // NUMAGENTS agents per hour
    static final double EXPNUMAGENTS = NUMAGENTS*(ENDFEEDBACK -
                                                STARTHOUR)*PERIODDURATION*5;

    // Contact center components
    PeriodChangeEvent pce; // Event marking new periods
    TrunkGroup trunks; // Manages phone lines
    PiecewiseConstantPoissonArrivalProcess donors;
    AgentGroup volunteers;
    WaitingQueue queue;
    Router router;

```

```

// Service time generator
ExponentialGen sgen = new ExponentialGen (new MRG32k3a(), 1.0/SERVICETIME);
// Patience time generator
ExponentialGen pgen = new ExponentialGen (new MRG32k3a(), 1.0/PATIENCETIME);

// Counters and probe used during replications
int numArrivals, numBlocked, numAbandoned,
    numMessages, numDisconnected, numGoodSL,
    numServed;
double sumWaitingTimes;
GroupVolumeStat vstat;

// Statistical collectors
Tally arrivals = new Tally ("Number of arrived contacts"),
    served = new Tally ("Number of served contacts"),
    blocked = new Tally ("Number of blocked contacts"),
    abandoned = new Tally ("Number of abandoned contacts"),
    messages = new Tally ("Number of messages"),
    disconnected = new Tally ("Number of disconnected contacts"),
    goodSL = new Tally ("Number of contacts in target"),
    serviceLevel = new Tally ("Service level"),
    occupancy = new Tally ("Agents' occupancy ratio");
FunctionOfMultipleMeansTally speedOfAnswer =
    new FunctionOfMultipleMeansTally
        (new RatioFunction(), "Speed of answer", 2);

Telethon() {
    pce = new PeriodChangeEvent (PERIODDURATION, P + 2, 0.0);
    final double[] arrivalRates = new double[P+2];
    for (int day = 0; day < 5; day++)
        for (int hour = STARTHOUR; hour < ENDDHOUR; hour++)
            arrivalRates[24*day + hour + 1] = ARRIVALRATE;
    donors = new PiecewiseConstantPoissonArrivalProcess
        (pce, new MyContactFactory(), arrivalRates, new MRG32k3a());
    donors.setNormalizing (true);
    trunks = new TrunkGroup (NUMTRUNKS);

    final int[] numAgents = new int[P+2];
    for (int day = 0; day < 5; day++)
        for (int hour = STARTHOUR; hour < ENDDHOUR; hour++)
            numAgents[24*day + hour + 1] = NUMAGENTS;
    volunteers = new AgentGroup (pce, numAgents);
    queue = new StandardWaitingQueue();
    queue.setMaximalQueueTimeGenerator
        (5, new ConstantValueGenerator (1, PATIENCETIME));
}

```

```

    router = createRouter();
    router.setClearWaitingQueues (true);
    router.setWaitingQueue (0, queue);
    router.setAgentGroup (0, volunteers);
    donors.addNewContactListener (router);
    router.addExitedContactListener (new MyContactMeasures());
    vstat = new GroupVolumeStat (volunteers);
}

Router createRouter() {
    return new QueuePriorityRouter (TYPETOGROUPMAP, GROUPTOTYPEMAP);
}

// Creates new contacts
class MyContactFactory implements ContactFactory {
    public Contact newInstance() {
        final Contact contact = new Contact();
        contact.setTrunkGroup (trunks);
        contact.setDefaultPatienceTime (pgen.nextDouble());
        contact.setDefaultServiceTime (sgen.nextDouble());
        return contact;
    }
}

// Updates counters when a contact exits
class MyContactMeasures implements ExitedContactListener {
    public void blocked (Router router, Contact contact, int bType) {
        ++numArrivals;    ++numBlocked;
    }

    public void dequeued (Router router, DequeueEvent ev) {
        ++numArrivals;
        if (ev.getEffectiveDequeueType() == 5)
            ++numMessages;
        else if (ev.getEffectiveDequeueType() == Router.DEQUEUEUETYPE_NOAGENT)
            ++numDisconnected;
        else
            ++numAbandoned;
    }

    public void served (Router router, EndServiceEvent ev) {
        ++numArrivals;    ++numServed;
        final double qt = ev.getContact().getTotalQueueTime();
        if (qt < AWT) ++numGoodSL;
        sumWaitingTimes += qt;
    }
}

```

```

public void simulateOneDay() {
    Sim.init();      pce.init();
    trunks.init();   donors.init();
    queue.init();    volunteers.init();
    numArrivals = numBlocked = numAbandoned
        = numMessages = numDisconnected = numGoodSL
        = numServed = 0;
    sumWaitingTimes = 0;
    vstat.init();
    donors.start();
    pce.start();
    Sim.start();
    pce.stop();
    addObs();
}

void addObs() {
    arrivals.add (numArrivals);
    served.add (numServed);
    goodSL.add (numGoodSL);
    serviceLevel.add (100*numGoodSL/EXPARRIVALS);
    final double Nb = vstat.getStatNumBusyAgents().sum();
    occupancy.add (100*Nb/EXPNUMAGENTS);
    blocked.add (numBlocked);
    abandoned.add (numAbandoned);
    messages.add (numMessages);
    disconnected.add (numDisconnected);
    speedOfAnswer.add (sumWaitingTimes, numServed);
}

void simulate (int days) {
    arrivals.init();      served.init();      blocked.init();
    abandoned.init();     messages.init();     disconnected.init();
    serviceLevel.init();  speedOfAnswer.init(); occupancy.init();
    goodSL.init();
    for (int r = 0; r < days; r++) simulateOneDay();
}

public void printStatistics() {
    System.out.println (arrivals.reportAndCISTudent (LEVEL, 3));
    System.out.println (served.reportAndCISTudent (LEVEL, 3));
    System.out.println (blocked.reportAndCISTudent (LEVEL, 3));
    System.out.println (abandoned.reportAndCISTudent (LEVEL, 3));
    System.out.println (messages.reportAndCISTudent (LEVEL, 3));
    System.out.println (disconnected.reportAndCISTudent (LEVEL, 3));
    System.out.println (goodSL.reportAndCISTudent (LEVEL, 3));
    System.out.println (speedOfAnswer.reportAndCIDelta (LEVEL, 3));
}

```



```

        System.out.println (serviceLevel.reportAndCISStudent (LEVEL, 3));
        System.out.println (occupancy.reportAndCISStudent (LEVEL, 3));
    }

    public static void main (String[] args) {
        final Telethon t = new Telethon();
        final Chrono timer = new Chrono();
        t.simulate (NUMDAYS);
        System.out.println ("CPU time: " + timer.format());
        t.printStatistics();
    }
}

```

Listing 18 gives the implementation of the simple non-stationary contact center described above. As with previous examples, fields are used for input data and objects, and a constructor is defined to create the necessary elements of the contact center. We first construct the event that will mark the beginning of new periods. In this example and all other sample programs adapted from Arena Contact Center Edition, the horizon is set to a full week, and simulation time is thus divided into $24 * 7$ 60-minutes main periods. The preliminary and wrap-up periods defined by the period-change event have length 0, but they must be counted in the total number of periods, so we need $24 * 7 + 2$ periods.

As usual, we then construct an arrival process using a contact factory, a random stream, and other parameters. In contrast with Arena Contact Center Edition only expecting the number of arrivals for each main period, ContactCenters requires the user to specify which arrival process to use along with its parameters. The normalization is enabled for the given rates to be divided by the period duration.

In contrast with most previous examples, the capacity of the system is limited. As a result, some donors may receive a busy signal and be blocked. It could be possible to limit the number of phone lines by setting the queue capacity to 12 as shown in section 1.4. With this setting, when the 12 agents are busy and 12 contacts are in queue, all the phone lines are busy, the system is full, and any new contact is blocked until a service ends or a donor leaves the queue. However, in general, because the number of agents in the system changes, the limit on queue capacity needs to be updated at the beginning of each period. It is also possible to associate some contact types, e.g., premium contacts using a specific phone bank as opposed to ordinary contacts accessing an independent pool of phone lines. Therefore, a more general framework for capacity limitation may be needed. The ContactCenters library supports the same notion of trunk groups as Arena Contact Center Edition, but it is disabled by default. A *trunk group* is a set of communication channels (e.g., phone lines) being allocated during the lifetime of contacts. For this facility to be enabled, each contact needs to be linked to a trunk group by using the `setTrunkGroup` method. A channel is then allocated when the contact reaches the router, and freed when it leaves the system. If a contact arrives when all channels of its trunk group are busy, it is blocked. To implement this feature in the Telethon example, the constructor creates a `TrunkGroup` instance which is associated with new contacts in the `newInstance` method of the contact factory.

The same constructor as in the two preceding examples is used to create the agent group and define the shift of the agents. In Arena Contact Center Edition, an agent group schedule defines the shifts for agents whereas agent groups only specify their capacity. The library does not define such an agent group schedule; agent groups can be created, as in section 1, with a fixed number of agents which can change at any time, or, as in sections 2 and 3, with a per-period number of members. In the example, the number of agents is constant, but it must be set to 0 when the contact center closes. Therefore, an array of $P + 2$ elements is constructed and any index corresponding to an opening hour is filled with 12, while the other values keep their default, 0.

We then construct a standard waiting queue to use a FIFO discipline. In Arena Contact Center Edition, each agent group is associated with a waiting queue. In the ContactCenters library, the waiting queues and agent groups are independent from each other for maximal flexibility. It is the router's task to create the connection between these two elements.

The same router as in sections 1 and 2 is used since there is a single contact type and a single agent group. However, to simulate the same model as with Arena Contact Center Edition, the queues have to be cleared when all the agents go off-duty. The `Router` class fortunately provides some facilities to perform this clearing automatically if it is activated by the `setClearWaitingQueues` method. For more simplicity, we do not simulate the 30-seconds delay necessary for a donor to leave the message before being disconnected. This aspect would require the definition of a custom router, which will be studied in the next subsection.

The `simulate` and `simulateOneDay` methods are very similar to their counterparts in previous examples. However, the simulation logic, i.e., which events are executed during `Sim.start`, differs slightly, because queued contacts can be disconnected.

Automatic queue clearing is implemented by the router while the disconnection after 2 minutes must be implemented by setting a secondary *queue time distribution* through the `setMaximalQueueTimeGenerator` method of `WaitingQueue`. By default, a waiting queue only has a primary maximal queue time generator associated with abandonment, i.e., dequeue type 1, and the generated value is the patience time extracted from contact objects. Each queue time distribution is given by an implementation of `ValueGenerator` which specifies a `nextDouble` method taking an argument of class `Contact` and returning a number. To support message leaving, the constructor of `Telethon` therefore creates a constant value generator for a single contact type and assigns it to dequeue type 5. This value was chosen arbitrarily, because any dequeue type different from 0 or 1 could be used.

When a contact enters the previously-configured queue, if the extracted patience time is greater than 2, the maximal queue time is set to 2, and the dequeue type is set to 5. If this maximal queue time is elapsed before a contact is served, the contact is disconnected, and is asked to leave a message. Otherwise, if the patience time is smaller than or equal to 2, the maximal queue time is the patience time and the dequeue type is 1. In this case, an abandonment occurs if the maximal queue time is elapsed before service starts.

As in previous examples, we use an exited-contact listener to count events. However, when abandonment, disconnection, and message leaving are considered simultaneously, counting

the dequeued contacts requires special care, because the `dequeued` method in `MyContactMeasures` receives various kinds of events. The effective dequeue type must therefore be used to differentiate the three types of removals. Dequeue type 1 is used for abandonment, type 5 is used for message leaving while the router defines its own dequeue type for disconnection when there is no agent. The dequeue type 0 is reserved for dequeued contacts to be served; these contacts are not notified to the `dequeued` method of `MyContactMeasures` since they have not left the contact center yet.

Statistical collecting is performed the same way as in the previous examples: during the simulation, values are counted and summed up. The waiting times of served contacts are also considered to get the average speed of answer. Another similar measure which is computed by the example in section 2 but not by this program is the average waiting time which includes the served and abandoned contacts. At the end of each replication, `addObs` is used to add the observations to the appropriate statistical collectors.

We can reduce the variance on the service level and the occupancy ratio by replacing the denominators with the true, known, expectations. The expected daily number of donors is $4 * 50 = 200$, so $\mathbb{E}[A] = 1000$ for the whole week, which excludes the weekend in this model. Since $N_0(t)$ is constant during all the opening hours, the integral of $N_0(t)$ is constant, because we never have ghost agents.

However, to get an accurate estimator of the occupancy ratio, we should not count the integral of busy agents during periods where $N_0(t) = 0$. We could achieve this using `ContactCenters` (see section 4.3), but the resulting occupancy ratio would not correspond to the output given by `Arena Contact Center Edition`.

Results of the simulation, presented on Listing 19, seem different from `Arena Contact Center Edition`, because different random seeds are used in the `ContactCenters` simulator. If the system is simulated with the same number of replications in both programs, the confidence intervals intersect, partly showing that the results are not significantly different. The simplification of the model to avoid a custom router does not seem to bias the estimators significantly.

Listing 19: Results of the program Telethon

```
CPU time: 0:0:1.2
REPORT on Tally stat. collector ==> Number of arrived contacts
  num. obs.      min      max      average      standard dev.
    1000      907.000    1108.000    1000.169        33.164
95.0% confidence interval for mean (student): ( 998.111, 1002.227 )

REPORT on Tally stat. collector ==> Number of served contacts
  num. obs.      min      max      average      standard dev.
    1000      890.000    1057.000     963.109        27.984
95.0% confidence interval for mean (student): ( 961.372, 964.846 )

REPORT on Tally stat. collector ==> Number of blocked contacts
  num. obs.      min      max      average      standard dev.
```

```

1000      0.000      0.000      0.000      0.000
95.0% confidence interval for mean (student): (    0.000,    0.000 )

REPORT on Tally stat. collector ==> Number of abandoned contacts
  num. obs.      min      max      average      standard dev.
    1000      4.000     67.000     31.820     10.437
95.0% confidence interval for mean (student): (    31.172,    32.468 )

REPORT on Tally stat. collector ==> Number of messages
  num. obs.      min      max      average      standard dev.
    1000      0.000     21.000      4.960      2.973
95.0% confidence interval for mean (student): (     4.776,     5.144 )

REPORT on Tally stat. collector ==> Number of disconnected contacts
  num. obs.      min      max      average      standard dev.
    1000      0.000      7.000      0.280      0.718
95.0% confidence interval for mean (student): (     0.235,     0.325 )

REPORT on Tally stat. collector ==> Number of contacts in target
  num. obs.      min      max      average      standard dev.
    1000     880.000    1042.000     950.265     27.173
95.0% confidence interval for mean (student): (    948.579,    951.951 )

REPORT on Tally stat. collector ==> Speed of answer
  func. of averages      standard dev.      num. obs.
        0.036          0.012          1000
95.0% confidence interval for function of means: (     0.036,     0.037 )

REPORT on Tally stat. collector ==> Service level
  num. obs.      min      max      average      standard dev.
    1000      88.000     104.200      95.026      2.717
95.0% confidence interval for mean (student): (    94.858,    95.195 )

REPORT on Tally stat. collector ==> Agents' occupancy ratio
  num. obs.      min      max      average      standard dev.
    1000      59.365     75.824     66.772      2.650
95.0% confidence interval for mean (student): (    66.607,    66.936 )

```

4.2 Modeling the message delay

After a contact waits for two minutes in the model of Arena Contact Center Edition, it leaves a message for a duration of 30 seconds before the phone line is released. This delay cannot be modeled by adding 0.5 minutes to the maximal queue time since an agent must not be able to pick a contact having waited more than two minutes while he is leaving its message. This aspect cannot be simulated unless the router is customized such as in Listing 20.

The `QueuePriorityRouter` must be extended to override its `dequeued` method being called every time a contact is removed from a waiting queue connected to the router. If the effective dequeue type is 0, the event must be ignored since the contact is about to be served. Otherwise, if the dequeue type is 5, an event is scheduled to happen after 30 seconds (0.5 minutes). This exiting event must be defined as an inner class in the custom router to call the protected `exitDequeued` method which releases the phone line and broadcasts the exited contact. If the dequeue type is 1, the `exitDequeued` method must be used for the phone line to be released at the time of abandonment. Note that this technique could also be used to model more realistic random message delays.

In this model, the delay is too short to have any impact on the results over all time. We may observe an effect on some individual periods, or if the arrival rates were higher.

Listing 20: Extension of Telethon for the 30 second delay

```
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.router.QueuePriorityRouter;
import umontreal.iro.lecuyer.contactcenters.router.Router;

import umontreal.iro.lecuyer.simevents.Event;
import umontreal.iro.lecuyer.util.Chrono;

public class TelethonMsg extends Telethon {
    @Override
    Router createRouter() {
        return new MyRouter (TYPETOGROUPMAP, GROUPTOTYPEMAP);
    }

    class MyRouter extends QueuePriorityRouter {
        MyRouter (int[][] gt, int[][] tg) { super (gt, tg); }
        @Override
        protected void dequeued (DequeueEvent ev) {
            final int dq = ev.getEffectiveDequeueType();
            if (dq == 0) return;
            else if (dq == 5) new MessageDelayEvent (ev).schedule (0.5);
            else exitDequeued (ev);
        }

        class MessageDelayEvent extends Event {
            DequeueEvent ev;
            MessageDelayEvent (DequeueEvent ev) {
                this.ev = ev;
            }
            @Override
            public void actions() { exitDequeued (ev); }
        }
    }
}
```

```

public static void main (String[] args) {
    final Telethon t = new TelethonMsg();
    final Chrono timer = new Chrono();
    t.simulate (NUMDAYS);
    System.out.println ("CPU time: " + timer.format());
    t.printStatistics();
}
}

```

4.3 Correcting the estimator for occupancy ratio

The integral for $N_{B,0}(t)$ needs to be computed for the opening hours only to get an unbiased estimator for the agents' occupancy ratio. The statistical collecting in `vstat` must be disabled when $N_0(t) = 0$ in order to achieve this.

In Listing 21, we define a new statistical collector for the corrected estimator as well as a new group-volume statistical counter. In the constructor, we call the superclass' constructor and connect a custom listener to the agent group before we create our new group volume statistical collector.

The custom listener, implemented in `MyAgentGroupListener`, reacts when the agent group is initialized, and the number of volunteers changes, i.e., goes from 0 to 12 or from 12 to 0. Setting the agent group of a group volume statistical counter modifies the list of listeners of the agent group. However, modifications to the list of listeners during the broadcast of an event can lead to unpredictable results. As a result, the program needs to create a support event that configures the group volume statistical collector. Moreover, to prevent inconsistencies, the group volume statistical collector needs to be initialized after the simulation clock is reset, which cannot be done without rewriting `simulateOneDay` or modifying the base class `Telethon`. Fortunately, when an agent group is initialized, it notifies every registered listener. We therefore react to this event by initializing the new group volume statistical counter.

When the number of agents goes from 12 to 0 at the end of the day, statistical collecting for `vstatCorr` needs to be disabled. This can be achieved by breaking the association between the agent group and the statistical counter. The internal counters then reset to 0 until the association is restored, at the beginning of the next day.

Listing 21: Extension of `Telethon` for corrected occupancy ratio estimator

```

import umontreal.iro.lecuyer.contactcenters.server.AgentGroup;
import umontreal.iro.lecuyer.contactcenters.server.AgentGroupListener;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;
import umontreal.iro.lecuyer.contactcenters.server.GroupVolumeStat;

import umontreal.iro.lecuyer.simevents.Event;

```

```
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.util.Chrono;

public class TelethonOcc extends Telethon {
    Tally occupancyCorr = new Tally ("Corrected agents' occupancy ratio");
    GroupVolumeStat vstatCorr;

    TelethonOcc() {
        super();
        volunteers.addAgentGroupListener (new MyAgentGroupListener());
        vstatCorr = new GroupVolumeStat (volunteers);
    }

    class MyAgentGroupListener implements AgentGroupListener {
        public void agentGroupChange (AgentGroup group) {
            if (group.getNumAgents() == 0)
                new SetAgentGroupEvent (null).scheduleNext();
            else
                new SetAgentGroupEvent (group).scheduleNext();
        }

        public void beginService (EndServiceEvent es) {}
        public void endContact (EndServiceEvent es) {}
        public void endService (EndServiceEvent es) {}
        public void init (AgentGroup group) { vstatCorr.init(); }
    }

    class SetAgentGroupEvent extends Event {
        AgentGroup group;

        public SetAgentGroupEvent (AgentGroup group) {
            this.group = group;
        }

        @Override
        public void actions() {
            vstatCorr.setAgentGroup (group);
        }
    }

    @Override
    public void addObs() {
        super.addObs();
        final double Nb = vstatCorr.getStatNumBusyAgents().sum();
        occupancyCorr.add (100.0*Nb/EXPNUMAGENTS);
    }
}
```

```

@Override
public void simulate (int n) {
    occupancyCorr.init();
    super.simulate (n);
}

@Override
public void printStatistics() {
    super.printStatistics();
    System.out.println (occupancyCorr.reportAndCISudent (LEVEL, 3));
}

public static void main (String[] args) {
    final Telethon t = new TelethonOcc();
    final Chrono timer = new Chrono();
    t.simulate (NUMDAYS);
    System.out.println ("CPU time: " + timer.format());
    t.printStatistics();
}
}

```

Listing 22 presents the output of the modified program. We see that the corrected occupancy ratio is significantly different from the ratio obtained using Arena Contact Center Edition.

Listing 22: Results of the program TelethonOcc

```

CPU time: 0:0:1.9
REPORT on Tally stat. collector ==> Number of arrived contacts
  num. obs.      min      max      average      standard dev.
    1000      907.000    1108.000    1000.169      33.164
95.0% confidence interval for mean (student): ( 998.111, 1002.227 )

REPORT on Tally stat. collector ==> Number of served contacts
  num. obs.      min      max      average      standard dev.
    1000      890.000    1057.000    963.109      27.984
95.0% confidence interval for mean (student): ( 961.372, 964.846 )

REPORT on Tally stat. collector ==> Number of blocked contacts
  num. obs.      min      max      average      standard dev.
    1000       0.000     0.000      0.000       0.000
95.0% confidence interval for mean (student): ( 0.000, 0.000 )

REPORT on Tally stat. collector ==> Number of abandoned contacts
  num. obs.      min      max      average      standard dev.
    1000       4.000     67.000     31.820     10.437

```


95.0% confidence interval for mean (student): (31.172, 32.468)

REPORT on Tally stat. collector ==> Number of messages

num. obs.	min	max	average	standard dev.
1000	0.000	21.000	4.960	2.973

95.0% confidence interval for mean (student): (4.776, 5.144)

REPORT on Tally stat. collector ==> Number of disconnected contacts

num. obs.	min	max	average	standard dev.
1000	0.000	7.000	0.280	0.718

95.0% confidence interval for mean (student): (0.235, 0.325)

REPORT on Tally stat. collector ==> Number of contacts in target

num. obs.	min	max	average	standard dev.
1000	880.000	1042.000	950.265	27.173

95.0% confidence interval for mean (student): (948.579, 951.951)

REPORT on Tally stat. collector ==> Speed of answer

func. of averages	standard dev.	num. obs.
0.036	0.012	1000

95.0% confidence interval for function of means: (0.036, 0.037)

REPORT on Tally stat. collector ==> Service level

num. obs.	min	max	average	standard dev.
1000	88.000	104.200	95.026	2.717

95.0% confidence interval for mean (student): (94.858, 95.195)

REPORT on Tally stat. collector ==> Agents' occupancy ratio

num. obs.	min	max	average	standard dev.
1000	59.365	75.824	66.772	2.650

95.0% confidence interval for mean (student): (66.607, 66.936)

REPORT on Tally stat. collector ==> Corrected agents' occupancy ratio

num. obs.	min	max	average	standard dev.
1000	57.250	72.489	64.049	2.499

95.0% confidence interval for mean (student): (63.894, 64.204)

5 Bilingual contact center

This example represents a contact center serving an English and a Spanish population. On each weekday, contacts of each type arrive following a Poisson process with rate 100/h from 8AM to 9AM, and 50/h from 9AM to 5PM. 7 English-speaking, 7 Spanish-speaking, and 4 bilingual agents are available to serve contacts. The contact center provides 40 phone lines to accommodate contacts.

When a contactor arrives, the router randomly selects a free agent or queues the contact if there is no free agent. Patience times are i.i.d. exponential variables with mean 2 minutes. I.i.d. service times follow the uniform distribution, ranging from 3 minutes to 7 minutes for English contacts, and from 4 minutes to 6 minutes for Spanish contacts.

If abandonment occurs, with probability 0.75, the contactor retries to join an agent after 20 minutes. Arena Contact Center Edition supports a generalization of this concept called *contact backs*. Whenever a contactor exits the system, he has the possibility to enter back with a certain probability, after a possibly random time. In the call center terminology, contact backs after abandonment are denoted *retrials* whereas contact backs after service are named *returns*. This differs from the concept of returns, i.e., agents recontacting customers, implemented in Arena Contact Center Edition.

The program estimates the service level for contact types $k = 0$ and 1, and the service level over all contact types, as well as the occupancy ratio for agent groups $i = 0, 1, 2$ and the ratio over all agent groups. Let $S_{G,k,\cdot}(s)$ be the number of served contacts of type k having waited in queue for less than s , and let $A_{k,\cdot}$ be the total number of arrivals of type k . The *service level for type k* is defined by

$$\frac{\mathbb{E}[S_{G,k,\cdot}(s)]}{\mathbb{E}[A_{k,\cdot}]}. \quad (8)$$

The occupancy ratio in group i is defined by (6). Since

$$\begin{aligned} S &= \sum_{k=0}^{K-1} S_{k,\cdot}, \\ A &= \sum_{k=0}^{K-1} A_{k,\cdot}, \\ N(t) &= \sum_{i=0}^{I-1} N_i(t), \\ N_B(t) &= \sum_{i=0}^{I-1} N_{B,i}(t), \\ &\text{etc.,} \end{aligned}$$

equations (7) and (3) can be used for the overall service level and occupancy ratio.

The program given on Listing 23 simulates the above model and computes required statistics.

Listing 23: A simulation program for a bilingual contact center

```

import umontreal.iro.lecuyer.contactcenters.ContactCenter;
import umontreal.iro.lecuyer.contactcenters.PeriodChangeEvent;
import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.contact.ContactFactory;
import umontreal.iro.lecuyer.contactcenters.contact.PiecewiseConstantPoissonArrivalProcess;
import umontreal.iro.lecuyer.contactcenters.contact.TrunkGroup;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.queue.StandardWaitingQueue;
import umontreal.iro.lecuyer.contactcenters.queue.WaitingQueue;
import umontreal.iro.lecuyer.contactcenters.router.AgentGroupSelectors;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.router.SingleFIFOQueueRouter;
import umontreal.iro.lecuyer.contactcenters.server.AgentGroup;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;
import umontreal.iro.lecuyer.contactcenters.server.GroupVolumeStat;

import umontreal.iro.lecuyer.probdist.UniformDist;
import umontreal.iro.lecuyer.randvar.ExponentialGen;
import umontreal.iro.lecuyer.randvar.UniformGen;
import umontreal.iro.lecuyer.rng.MRG32k3a;
import umontreal.iro.lecuyer.rng.RandomStream;
import umontreal.iro.lecuyer.simevents.Event;
import umontreal.iro.lecuyer.simevents.Sim;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.stat.list.ListOfTallies;
import umontreal.iro.lecuyer.util.Chrono;

public class Bilingual {
    // Contact type and agent group identifiers and names
    static final int ENGLISH      = 0;
    static final int SPANISH      = 1;
    static final int BILINGUAL    = 2;
    // TYPENAMES[k] gives the name of contact type k, for k = 0 and 1
    // TYPENAMES[K] gives the name for all contact types
    static final String[] TYPENAMES = {
        "English", "Spanish","all types"
    };
    // GROUPNAMES[i] gives the name of agent group i, for i = 0, 1, and 2
    // GROUPNAMES[I] gives the name for all agent groups
    static final String[] GROUPNAMES = {
        "English-only", "Spanish-only", "bilingual", "all groups"
    };

    // Input data (times are in minutes)
    static final int K            = 2;

```

```

static final int I                = 3;
static final int P                = 24*7;
static final double PERIODDURATION = 60;
static final int STARTHOUR        = 8;
static final int ENDFEEDBACK      = 17;
static final double FIRSTHOURARRIVALRATE = 100;
static final double ARRIVALRATE = 50;
// Capacity of each agent group
// 7 English-only, 7 Spanish-only, 4 bilingual
static final int[] NUMAGENTS = { 7, 7, 4 };
static final int NUMTRUNKS   = 40;
static final double AWT      = 1.0;
static final double PTIME    = 2.0;
static final double STIMEENGLISHMIN = 3.0;
static final double STIMEENGLISHMAX = 7.0;
static final double STIMESPANISHMIN = 4.0;
static final double STIMESPANISHMAX = 6.0;
static final double PROBCBACK   = 0.75;
static final double CBACKTIME   = 20.0;
static final int[][] TYPETOGROUPMAP = {
    // English contact type
    { ENGLISH, BILINGUAL },
    // Spanish contact type
    { SPANISH, BILINGUAL },
};
static final int[][] GROUPTOTYPEMAP = {
    // English-only agents
    { ENGLISH },
    // Spanish-only agents
    { SPANISH },
    // Bilingual agents
    { ENGLISH, SPANISH }
};

static final double LEVEL          = 0.95;    // Level of conf. int.
static final int NUMDAYS           = 1000;

// Known expectations
static final double EXPARRIVALS = 5*(FIRSTHOURARRIVALRATE +
                                     (ENDFEEDBACK - STARTHOUR - 1)*ARRIVALRATE);
static double[] EXPNUMAGENTS = new double[I + 1];
static {
    for (int i = 0; i < I; i++) {
        EXPNUMAGENTS[i] = 5*PERIODDURATION*(ENDFEEDBACK - STARTHOUR)*NUMAGENTS[i];
        EXPNUMAGENTS[I] += EXPNUMAGENTS[i];
    }
}

```

```

// Contact center components
PeriodChangeEvent pce;    // Event marking new periods
TrunkGroup trunks;       // Manages phone lines
PiecewiseConstantPoissonArrivalProcess[] arrivProc = new
    PiecewiseConstantPoissonArrivalProcess[K];
AgentGroup[] agentGroups = new AgentGroup[I];
WaitingQueue[] queues = new WaitingQueue[K];
Router router;

// Random streams and random variate generators
RandomStream agentStream = new MRG32k3a();
RandomStream streamContactBack = new MRG32k3a();
// Patience time generator
ExponentialGen pgen = new ExponentialGen (new MRG32k3a(), 1.0/PTIME);
// Service time generator, for each contact type
UniformGen[] sgen      = {
    // English
    new UniformGen (new MRG32k3a(), new UniformDist
        (STIMEENGLISHMIN, STIMEENGLISHMAX)),
    // Spanish
    new UniformGen (new MRG32k3a(), new UniformDist
        (STIMESPANISHMIN, STIMESPANISHMAX))
};

// Counters and probes used during replications
double[] numArriv  = new double[K + 1];
double[] numContactBack = new double[K + 1];
double[] numGoodSL = new double[K + 1];
double[] numServed = new double[K + 1];
double[] numDisconnected = new double[K + 1];
double[] numAbandoned = new double[K + 1];
GroupVolumeStat[] vstat = new GroupVolumeStat[I];
// Used as a temporary variable in addObs
double[] Nb = new double[I + 1];

// Statistical collectors
ListOfTallies<Tally> arrivals = create
    ("Number of arrived contacts", TYPENAMES);
ListOfTallies<Tally> contactBack = create
    ("Number of retrials after abandonment", TYPENAMES);
ListOfTallies<Tally> served = create
    ("Number of served contacts", TYPENAMES);
ListOfTallies<Tally> disconnected = create
    ("Number of disconnected contacts", TYPENAMES);
ListOfTallies<Tally> abandoned = create
    ("Number of abandoned contacts", TYPENAMES);

```

```

ListOfTallies<Tally> goodSL = create
    ("Number of contacts in target", TYPENAMES);
ListOfTallies<Tally> serviceLevel = create
    ("Service level", TYPENAMES);
ListOfTallies<Tally> occupancy = create
    ("Agents' occupancy ratio", GROUPNAMES);

private ListOfTallies<Tally> create (String name, String[] elementNames) {
    final ListOfTallies<Tally> lt = ListOfTallies.createWithTally (elementNames.length);
    lt.setName (name);
    for (int i = 0; i < elementNames.length; i++) {
        lt.get (i).setName (elementNames[i]);
        lt.get (i).setConfidenceIntervalStudent();
        lt.get (i).setConfidenceLevel (LEVEL);
    }
    return lt;
}

Bilingual() {
    pce = new PeriodChangeEvent (PERIODDURATION, P + 2, 0.0);
    trunks = new TrunkGroup (NUMTRUNKS);

    final double[] arrivalRates = new double[P + 2];
    for (int day = 0; day < 5; day++) {
        arrivalRates[24*day + STARTHOUR + 1] = FIRSTHOURLARRIVALRATE;
        for (int hour = STARTHOUR + 1; hour < ENDOUR; hour++)
            arrivalRates[24*day + hour + 1] = ARRIVALRATE;
    }
    for (int k = 0; k < K; k++) {
        arrivProc[k] = new PiecewiseConstantPoissonArrivalProcess
            (pce, new MyContactFactory (k), arrivalRates, new MRG32k3a());
        arrivProc[k].setNormalizing (true);
    }

    final int[][] numAgents = new int[I][P + 2];
    for (int day = 0; day < 5; day++)
        for (int hour = STARTHOUR; hour < ENDOUR; hour++) {
            final int ind = 24*day + hour + 1;
            for (int i = 0; i < I; i++)
                numAgents[i][ind] = NUMAGENTS[i];
        }
    for (int i = 0; i < I; i++) {
        agentGroups[i] = new AgentGroup (pce, numAgents[i]);
        vstat[i] = new GroupVolumeStat (agentGroups[i]);
    }
    for (int q = 0; q < K; q++)
        queues[q] = new StandardWaitingQueue();
}

```

```

    router = new MyRouter (TYPETOGROUPMAP, GROUPTOTYPEMAP);
    for (int k = 0; k < K; k++) arrivProc[k].addNewContactListener (router);
    for (int q = 0; q < K; q++) router.setWaitingQueue (q, queues[q]);
    for (int i = 0; i < I; i++) router.setAgentGroup (i, agentGroups[i]);
    router.setClearWaitingQueues (true);
    router.addExitedContactListener (new MyContactMeasures());
}

// Add a flag for determining if the contacts have retried
// after abandoning.
class MyContact extends Contact {
    boolean contactBack = false;
    MyContact (int type) {
        super (type);
    }
}

// Creates new contacts
class MyContactFactory implements ContactFactory {
    int type;
    MyContactFactory (int type) { this.type = type; }

    public Contact newInstance() {
        final MyContact contact = new MyContact (type);
        contact.setTrunkGroup (trunks);
        contact.setDefaultPatienceTime (pgen.nextDouble());
        contact.setDefaultServiceTime (sgen[type].nextDouble());
        return contact;
    }
}

// Implements random agent selection and retrials.
class MyRouter extends SingleFIFOQueueRouter {
    MyRouter (int[][] typeToGroupMap, int[][] groupToTypeMap) {
        super (typeToGroupMap, groupToTypeMap);
    }

    // Random agent selection
    @Override
    protected EndServiceEvent selectAgent (Contact contact) {
        final int tid = contact.getTypeId();
        final AgentGroup group = AgentGroupSelectors.selectUniform
            (this, typeToGroupMap[tid], agentStream);
        if (group == null) return null;
        return group.serve (contact);
    }
}

```

```

// Retrial with some probability for abandoned contacts
@Override
protected void dequeued (DequeueEvent ev) {
    if (ev.getEffectiveDequeueType() == 0) return;
    if (ev.getEffectiveDequeueType() == 1) {
        final double u = streamContactBack.nextDouble();
        if (u <= PROBCBACK)
            new ContactBackEvent (ev.getContact()).schedule (CBACKTIME);
    }
    exitDequeued (ev);
}
}

// Event happening when a customer contacts back
class ContactBackEvent extends Event {
    Contact contact;
    ContactBackEvent (Contact contact) { this.contact = contact; }

    @Override
    public void actions() {
        contact.setExited (false);
        ((MyContact)contact).contactBack = true;
        router.newContact (contact);
    }
}

// Updates counters
class MyContactMeasures implements ExitedContactListener {
    public void blocked (Router router, Contact contact, int bType) {
        countArrival (contact);
    }

    private void countArrival (Contact contact) {
        final int type = contact.getTypeId();
        ++numArriv[type];
        ++numArriv[K];
        if (((MyContact)contact).contactBack) {
            ++numContactBack[type];
            ++numContactBack[K];
        }
    }
}

public void dequeued (Router router, DequeueEvent ev) {
    final Contact contact = ev.getContact();
    countArrival (contact);
    final int type = contact.getTypeId();

```



```

        if (ev.getEffectiveDequeueType() == Router.DEQUEUEUETYPE_NOAGENT) {
            ++numDisconnected[type];
            ++numDisconnected[K];
        }
        else {
            ++numAbandoned[type];
            ++numAbandoned[K];
        }
    }
}

public void served (Router router, EndServiceEvent ev) {
    final Contact contact = ev.getContact();
    countArrival (contact);
    final int type = contact.getTypeId();
    ++numServed[type];
    ++numServed[K];
    if (contact.getTotalQueueTime() < AWT) {
        ++numGoodSL[type];
        ++numGoodSL[K];
    }
}
}

void simulateOneDay() {
    Sim.init();
    // Initializes period-change event, trunk group, arrival processes,
    // waiting queues, and agent groups.
    pce.init();
    trunks.init();
    ContactCenter.initElements (arrivProc);
    ContactCenter.initElements (agentGroups);
    ContactCenter.initElements (queues);
    for (int k = 0; k <= K; k++) {
        numArriv[k] = 0;
        numServed[k] = 0;
        numGoodSL[k] = 0;
        numDisconnected[k] = 0;
        numAbandoned[k] = 0;
        numContactBack[k] = 0;
    }
    for (final GroupVolumeStat element : vstat)
        element.init();
    ContactCenter.toggleElements (arrivProc, true);
    pce.start();
    Sim.start();
    pce.stop();
    addObs();
}

```

```

}

protected void addObs() {
    arrivals.add (numArriv);
    contactBack.add (numContactBack);
    served.add (numServed);
    disconnected.add (numDisconnected);
    abandoned.add (numAbandoned);
    goodSL.add (numGoodSL);
    for (int k = 0; k < K; k++)
        numGoodSL[k] /= EXPARRIVALS/100.0;
    numGoodSL[K] /= K*EXPARRIVALS/100.0;
    serviceLevel.add (numGoodSL);
    Nb[I] = 0;
    for (int i = 0; i < I; i++) {
        Nb[i] = vstat[i].getStatNumBusyAgents().sum();
        Nb[I] += Nb[i];
        Nb[i] /= EXPNUMAGENTS[i]/100.0;
    }
    Nb[I] /= EXPNUMAGENTS[I]/100.0;
    occupancy.add (Nb);
}

void simulate (int days) {
    arrivals.init();      served.init();      disconnected.init();
    abandoned.init();     serviceLevel.init(); occupancy.init();
    goodSL.init();        contactBack.init();
    for (int r = 0; r < days; r++) simulateOneDay();
}

public void printStatistics() {
    System.out.println (arrivals.report ());
    System.out.println (contactBack.report ());
    System.out.println (served.report ());
    System.out.println (disconnected.report ());
    System.out.println (abandoned.report ());
    System.out.println (goodSL.report ());
    System.out.println (serviceLevel.report ());
    System.out.println (occupancy.report ());
}

public static void main (String[] args) {
    final Bilingual b = new Bilingual();
    final Chrono timer = new Chrono();
    b.simulate (NUMDAYS);
    System.out.println ("CPU time: " + timer.format());
    b.printStatistics();
}

```

```

}
}

```

The program uses a helper class called **ContactCenter** providing convenience static methods for initializing components of the contact center.

In real-life models, contact types and agent groups have names. However, for efficiency, to distinguish contact types and agent groups, the library uses numerical identifiers rather than strings. On the other hand, replacing names with numerical identifiers reduces the readability of the program and the produced statistical reports. To make the program clearer, constants are associated with each identifier. The arrays **TYPENAMES** and **GROUPNAMES** contain names for contact types and agent groups being used for statistical reports.

In this model, we are interested in statistics for each contact type separately as well as for all contact types. In the program, many scalars are then replaced by arrays of $K+1$ elements, and lists of tallies are used instead of tallies for easier statistical collecting and reporting. Each list of tally has a *global name* corresponding to a type of performance measure, e.g., service level. Each element in a list of tallies also has its own *local name* corresponding to a particular measure, e.g., the service level for a contact type, or the occupancy ratio of the agents in a a group. The list of tallies are constructed using a **create** method that also initializes the global and local names, and set reporting options to show confidence intervals.

The model in Arena Contact Center Edition is made of two parent groups which create two waiting queues for contacts. A *parent group* is a set of agent groups defining a selection rule for the agents and assigning a preference for each contact type. To implement the parent group efficiently, a queue is created for each contact type and the contact selection is implemented in the router.

This contact center simulator requires a custom router to implement the random selection of agents as well as the contact back mechanism. This router, represented by the **MyRouter** class, extends **SingleFIFOQueueRouter** used in section 3. For this model, the type-to-group map indicates that an English contact must be served by an English-speaking or a bilingual agent, in that order. A Spanish contact must be served by a Spanish-speaking or a bilingual agent. The group-to-type map assigns contact types to agents: an English-speaking agent can serve English people only whereas a Spanish-speaking agent can serve Spanish people. The bilingual agents are generalists since they can serve any contact type in this system.

The agent selection rule of the chosen router is not appropriate, since we do not want specialists to have priority over generalists in this particular example. To implement random agent selection, the **selectAgent** method is thus overridden. This method is called by the router when a contact arrives and must be assigned an agent. It selects an agent, starts the service, and returns a reference to the end-service event. The **selectUniform** helper method manages the random selection automatically: each agent group is assigned a probability of selection given by the number of free agents in the group over the total number of free agents capable of serving the contact. If no group contains free agents, a **null** agent group reference is returned by **selectUniform**, and **selectAgent** returns **null** to indicate the router that the contact must be queued. Otherwise, the service of the contact is started.

When an agent becomes free, the router tries to pull contacts from waiting queues. In this example, the contact selection rule of the inherited router does not need to be customized. For specialists, the appropriate waiting queue is checked for a new contact which is removed and served. For generalists, both waiting queues are checked and the contact with the longest waiting time is removed.

Further customization is needed to support retrials, by overriding the `dequeued` method in the router. First, if the effective dequeue type is 0, the method returns immediately because this is not an abandoning contact; the dequeue type 0 is reserved for contacts being removed from the queue to be served. If the dequeue type is 1, with some probability, a `ContactBackEvent` is scheduled after a fixed delay. The contact then leaves the system, using the `exitDequeued` method. It is important to remove the contact from the system in order to free its associated phone line. When the contact-back event occurs, the `newContact` method is called with the `Contact` object. This is exactly what is performed by an arrival process broadcasting a new contact to the registered router.

However, some internal checks are performed to avoid contacts entering multiple times in the system, which would produce wrong simulation results. When a contact exits the system, the router sets an internal exit indicator that must be cleared before the contact can enter the router again.

To count the number of retrials, we need to define a subclass of `Contact` with a new field being updated when retrial occurs. The subclass `MyContact` must be referred to in `MyContactFactory` to get the appropriate instances and typecasts are often needed in listeners such as `MyContactMeasures`.

As we can see in the program, using arrays requires more work than scalars. Of course, they must be constructed when declared or in the constructor of the simulator. Then, at the beginning of each replication, in the `simulateOneDay` method, each element must be initialized separately. Finally, in the exited-contact listener, events must be counted in two elements of the arrays. When the event concerns a contact of type k , the index k of the arrays must be updated in addition to index K for the measures for all types. In addition, in `addObs`, estimating the occupancy ratio requires $\int_0^T N_{B,i}(t) dt$ and $\int_0^T N_i(t) dt$ to be obtained for each agent group separately.

`report` is used to format reports about each contact type or agent group, depending on the type of performance measure. The list report contains the same information as the ordinary report, but, as shown on Listing 24, it is more condensed and more appropriate for related performance measures.

Listing 24: Results of the program Bilingual

CPU time: 0:0:4.10						
Report for Number of arrived contacts						
	num obs.	min	max	average	std. dev.	conf. int.
English	1000	2369.000	2745.000	2555.811	56.822	95.0% (2552.285, 2559.337)
Spanish	1000	2398.000	2743.000	2554.350	57.003	95.0% (2550.813, 2557.887)
all types	1000	4826.000	5372.000	5110.161	83.029	95.0% (5105.009, 5115.313)

Report for Number of retrials after abandonment

	num obs.	min	max	average	std. dev.	conf. int.	
English	1000	17.000	111.000	54.073	13.495	95.0% (53.236,	54.910)
Spanish	1000	9.000	110.000	53.095	13.253	95.0% (52.273,	53.917)
all types	1000	55.000	187.000	107.168	21.377	95.0% (105.841,	108.495)

Report for Number of served contacts

	num obs.	min	max	average	std. dev.	conf. int.	
English	1000	2330.000	2650.000	2483.933	48.612	95.0% (2480.916,	2486.950)
Spanish	1000	2344.000	2651.000	2483.302	49.449	95.0% (2480.233,	2486.371)
all types	1000	4716.000	5192.000	4967.235	69.111	95.0% (4962.946,	4971.524)

Report for Number of disconnected contacts

	num obs.	min	max	average	std. dev.	conf. int.	
English	1000	0.000	3.000	0.030	0.222	95.0% (0.016,	0.044)
Spanish	1000	0.000	2.000	0.028	0.193	95.0% (0.016,	0.040)
all types	1000	0.000	3.000	0.058	0.291	95.0% (0.040,	0.076)

Report for Number of abandoned contacts

	num obs.	min	max	average	std. dev.	conf. int.	
English	1000	31.000	142.000	71.848	16.652	95.0% (70.815,	72.881)
Spanish	1000	17.000	137.000	71.020	16.459	95.0% (69.999,	72.041)
all types	1000	73.000	256.000	142.868	26.740	95.0% (141.209,	144.527)

Report for Number of contacts in target

	num obs.	min	max	average	std. dev.	conf. int.	
English	1000	2313.000	2613.000	2448.662	46.470	95.0% (2445.778,	2451.546)
Spanish	1000	2306.000	2615.000	2448.223	47.889	95.0% (2445.251,	2451.195)
all types	1000	4658.000	5104.000	4896.885	66.002	95.0% (4892.789,	4900.981)

Report for Service level

	num obs.	min	max	average	std. dev.	conf. int.	
English	1000	92.520	104.520	97.946	1.859	95.0% (97.831,	98.062)
Spanish	1000	92.240	104.600	97.929	1.916	95.0% (97.810,	98.048)
all types	1000	93.160	102.080	97.938	1.320	95.0% (97.856,	98.020)

Report for Agents' occupancy ratio

	num obs.	min	max	average	std. dev.	conf. int.	
English-only	1000	44.783	50.922	47.624	0.977	95.0% (47.563,	47.684)
Spanish-only	1000	44.830	50.927	47.639	0.962	95.0% (47.580,	47.699)
bilingual	1000	59.156	65.900	63.254	0.908	95.0% (63.197,	63.310)
all groups	1000	48.433	53.713	51.103	0.719	95.0% (51.059,	51.148)

6 Bank model

This example represents a bank model where each agent can process all contact types, but agents serve their specialty more efficiently. The bank can receive contacts for Savings, Checking, and Account Balance, and employs 4 Checking specialists and 3 Savings specialists. The Checking and Savings contacts are preferred to be handled by specialists when possible, because service times are multiplied by a 0.75 reward factor. For the Account Balance contacts, the system must use the two agent groups in a balanced way, so agent selection is random and uniform among free agents. For this model, patience times are uniform and service times are exponential. Arrivals follow a non-homogeneous Poisson process whose rate depends on the period of the day and the contact type. Table 1 presents the rest of the input data for this model.

Table 1: Input data for the Bank contact center

	Savings	Checking	Account balance
Patience	$U(3\text{min}, 5\text{min})$	$U(3\text{min}, 5\text{min})$	$U(2.5\text{min}, 3.5\text{min})$
Service	$\text{Exp}(5\text{min})$	$\text{Exp}(5\text{min})$	$\text{Exp}(1\text{min})$
Awt	60s	90s	30s
Arrivals from 8AM to 9AM	4/h	40/h	100/h
Hourly arrivals from 9AM to 5PM	2/h	20/h	50/h

The program estimates the service level as well as the occupancy ratio, defined by (8) and (6), respectively. The program also estimates per-contact type occupancy ratio defined by

$$o_{i,k} = \frac{E \left[\int_0^T N_{B,i,k}(t) dt \right]}{E \left[\int_0^T (N_i(t) + N_{G,i}(t)) dt \right]} \quad (9)$$

for $i = 0, \dots, I - 1$, and the global contact-type specific occupancy ratio

$$o_{I,k} = \frac{E \left[\int_0^T \sum_{i=0}^{I-1} N_{B,i,k}(t) dt \right]}{E \left[\int_0^T \sum_{i=1}^{I-1} (N_i(t) + N_{G,i}(t)) dt \right]}.$$

which is not estimated by Arena Contact Center Edition.

In the Arena Contact Center Edition model, idle and busy costs are assigned to the two agent groups. The expected busy cost of agents in group i $C_{b,i}$ is defined by

$$C_{b,i} = c_{b,i} E \left[\int_0^T N_{B,i}(t) dt \right],$$

where $c_{b,i}$ is the cost of one busy agent in group i per simulation time unit. Similarly, the idle cost of agents in group i is defined by

$$C_{i,i} = c_{i,i} E \left[\int_0^T N_{I,i}(t) dt \right].$$

The cost of Savings specialists is 12\$/hour while the cost of Checking specialists is 7.50\$/hour, whether the agents are idle or busy. Of course, more general costing models could be implemented by a ContactCenters model, e.g., time-varying idle and busy costs.

Listing 25 gives the implementation of the above model to estimate various performance measures for each contact type as well as aggregate measures for all types.

Listing 25: A simulation program for a bank model

```
import java.util.ArrayList;
import java.util.List;

import umontreal.iro.lecuyer.contactcenters.ContactCenter;
import umontreal.iro.lecuyer.contactcenters.MatrixUtil;
import umontreal.iro.lecuyer.contactcenters.NonStationaryMeasureMatrix;
import umontreal.iro.lecuyer.contactcenters.PeriodChangeEvent;
import umontreal.iro.lecuyer.contactcenters.RepSimCC;
import umontreal.iro.lecuyer.contactcenters.StatUtil;
import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.contact.ContactFactory;
import umontreal.iro.lecuyer.contactcenters.contact.ContactSumMatrix;
import umontreal.iro.lecuyer.contactcenters.contact.PiecewiseConstantPoissonArrivalProcess;
import umontreal.iro.lecuyer.contactcenters.contact.TrunkGroup;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.queue.StandardWaitingQueue;
import umontreal.iro.lecuyer.contactcenters.queue.WaitingQueue;
import umontreal.iro.lecuyer.contactcenters.router.AgentGroupSelectors;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.router.SingleFIFOQueueRouter;
import umontreal.iro.lecuyer.contactcenters.server.Agent;
import umontreal.iro.lecuyer.contactcenters.server.AgentGroup;
import umontreal.iro.lecuyer.contactcenters.server.ContactTimeGenerator;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;
import umontreal.iro.lecuyer.contactcenters.server.GroupVolumeStatMeasureMatrix;
import umontreal.iro.lecuyer.probdist.UniformDist;
import umontreal.iro.lecuyer.randvar.ExponentialGen;
import umontreal.iro.lecuyer.randvar.UniformGen;
import umontreal.iro.lecuyer.rng.MRG32k3a;
import umontreal.iro.lecuyer.rng.RandomStream;
import umontreal.iro.lecuyer.simevents.Sim;
import umontreal.iro.lecuyer.stat.FunctionOfMultipleMeansTally;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.stat.matrix.MatrixOfFunctionOfMultipleMeansTallies;
import umontreal.iro.lecuyer.stat.matrix.MatrixOfTallies;
import umontreal.iro.lecuyer.stat.mperiods.IntegralMeasureMatrix;
import umontreal.iro.lecuyer.stat.mperiods.MeasureMatrix;
import umontreal.iro.lecuyer.stat.mperiods.MeasureSet;
```

```

import umontreal.iro.lecuyer.util.Chrono;
import umontreal.iro.lecuyer.util.RatioFunction;
import cern.colt.matrix.DoubleMatrix2D;
import cern.jet.math.Functions;

public class Bank {
    // Contact type and agent group identifiers and names
    static final int SAVINGS          = 0;
    static final int CHECKING         = 1;
    static final int BALANCE          = 2;
    static final String[] TYPENAMES   = {
        "savings", "checking", "account balance", "all types"
    };
    static final String[] GROUPNAMES  = {
        "savings specialists", "checking specialists", "all groups"
    };
    static final int K                = 3;
    static final int I                = 2;
    static final int P                = 24*7;
    static final String[] GROUPTYPENAMES = new String[K*(I + 1)];
    // PERIODNAMES[p] gives the name for period p
    static final String[] PERIODNAMES  = new String[P + 1];

    static {
        for (int i = 0; i <= I; i++)
            for (int k = 0; k < K; k++) {
                String tn = TYPENAMES[k];
                String gn = GROUPNAMES[i];
                if (tn.equals("account balance"))
                    tn = "balance";
                gn = gn.replaceFirst("specialists", "spec.");
                GROUPTYPENAMES[i*K + k] = gn + "/" + tn;
            }
        for (int p = 0; p < P; p++)
            PERIODNAMES[p] = "period " + (p + 1);
        PERIODNAMES[P] = "all periods";
    }

    // Input data (all times are in minutes)
    // Cost of agents per minute.
    // The array contains the cost of Savings and Checking specialists,
    // in that order.
    static final double[] COST          = { 12.0/60, 7.5/60 };
    static final double PERIODDURATION = 60.0;
    static final int STARTHOUR          = 8;
    static final int ENDOUR             = 17;
    static final int NUMTRUNKS          = 15;

```



```

static final double[] FIRSTHOURARRIVALRATES =
{ 4, 40, 100 };
static final double[] ARRIVALRATES = { 2, 20, 50 };
static final int[] NUMAGENTS = { 3, 4 };
// Service time multipliers
static final double[][] STIMEMULT = {
    // Savings specialists
    { 0.75, 1.0, 1.0 },
    // Checking specialists
    { 1.0, 0.75, 1.0 }
};

// Mean service times
static final double[] STIME          = { 5, 5, 1 };
static final double[] AWT            = { 1.0, 90.0/60.0, 30.0/60.0 };
static final int[][] TYPETOGROUPMAP = {
    // Savings contact type
    { SAVINGS, CHECKING },
    // Checking contact type
    { CHECKING, SAVINGS },
    // Account balance contact type
    { SAVINGS, CHECKING }
};
static final int[][] GROUPTOTYPEMAP = {
    // Savings specialists
    { SAVINGS, CHECKING, BALANCE },
    // Checking specialists
    { CHECKING, SAVINGS, BALANCE }
};
static final double LEVEL            = 0.95; // Level of conf. int.
static final int NUMDAYS              = 1000;

// Contact center components
PeriodChangeEvent pce; // Event marking beginning of periods
TrunkGroup trunks;    // Manages phone lines
PiecewiseConstantPoissonArrivalProcess[] arrivProc =
    new PiecewiseConstantPoissonArrivalProcess[K];
AgentGroup[] agentGroups = new AgentGroup[I];
WaitingQueue[] queues = new WaitingQueue[K];
Router router;

// Random streams and random variate generators
RandomStream agentStream = new MRG32k3a();
UniformGen[] pgen = {
    // Savings
    new UniformGen (new MRG32k3a(), new UniformDist (3.0, 5.0)),
    // Checking

```

```

    new UniformGen (new MRG32k3a(), new UniformDist (3.0, 5.0)),
    // Account balance
    new UniformGen (new MRG32k3a(), new UniformDist (2.5, 3.5))
};
ExponentialGen sgen[] = {
    // Savings
    new ExponentialGen (new MRG32k3a(), 1.0/STIME[SAVINGS]),
    // Checking
    new ExponentialGen (new MRG32k3a(), 1.0/STIME[CHECKING]),
    new ExponentialGen (new MRG32k3a(), 1.0/STIME[BALANCE])
};

// Measure matrices (counters)
List<MeasureMatrix> mats = new ArrayList<MeasureMatrix>();
ContactSumMatrix numArriv;
ContactSumMatrix numBlocked;
ContactSumMatrix numGoodSL;
ContactSumMatrix numServed;
ContactSumMatrix numAbandoned;
ContactSumMatrix numDisconnected;
ContactSumMatrix sumServiceTimes;
ContactSumMatrix sumWaitingTimes;
IntegralMeasureMatrix<?>[] vstat = new IntegralMeasureMatrix[I];
MeasureSet svm;
MeasureSet ivm;
MeasureSet tvn;
MeasureSet dsvm;
MeasureSet dtvm;

// Statistical collectors
List<MatrixOfTallies<Tally>> mTallies = new ArrayList<MatrixOfTallies<Tally>>();
MatrixOfTallies<Tally> arrivals;
MatrixOfTallies<Tally> blocked;
MatrixOfTallies<Tally> served;
MatrixOfFunctionOfMultipleMeansTallies<FunctionOfMultipleMeansTally> serviceLevel;
MatrixOfTallies<Tally> occupancy;
MatrixOfTallies<Tally> occupancyPerType;
MatrixOfTallies<Tally> serviceTimes;
MatrixOfTallies<Tally> idleCost;
MatrixOfTallies<Tally> busyCost;
MatrixOfTallies<Tally> waitingTimes;
MatrixOfTallies<Tally> disconnected;
MatrixOfTallies<Tally> abandoned;
Tally trunkAvail;
Tally trunkBusy;

Bank() {

```

```

pce = new PeriodChangeEvent (PERIODDURATION, P + 2, 0.0);
trunks = new TrunkGroup (NUMTRUNKS);
trunks.setStatCollecting (true);

final double[][] arrivalRates = new double[K][P + 2];
for (int day = 0; day < 5; day++) {
    for (int k = 0; k < K; k++)
        arrivalRates[k][24*day + STARTHOUR + 1] =
            FIRSTHOURARRIVALRATES[k];
    for (int hour = STARTHOUR + 1; hour < ENDFHOUR; hour++)
        for (int k = 0; k < K; k++)
            arrivalRates[k][24*day + hour + 1] = ARRIVALRATES[k];
}
for (int k = 0; k < K; k++) {
    arrivProc[k] = new PiecewiseConstantPoissonArrivalProcess
        (pce, new MyContactFactory (k), arrivalRates[k], new MRG32k3a());
    arrivProc[k].setNormalizing (true);
}

final int[][] numAgents = new int[I][P + 2];
for (int day = 0; day < 5; day++)
    for (int hour = STARTHOUR; hour < ENDFHOUR; hour++)
        for (int i = 0; i < I; i++)
            numAgents[i][24*day + hour + 1] = NUMAGENTS[i];
for (int i = 0; i < I; i++) {
    agentGroups[i] = new AgentGroup (pce, numAgents[i]);
    agentGroups[i].setContactTimeGenerator
        (0, new ContactTimeGenerator (agentGroups[i], STIMEMULT[i]));
}

for (int q = 0; q < K; q++)
    queues[q] = new StandardWaitingQueue();

router = new MyRouter (TYPETOGROUPMAP, GROUPTOTYPEMAP);
for (int k = 0; k < K; k++) arrivProc[k].addNewContactListener (router);
for (int q = 0; q < K; q++) router.setWaitingQueue (q, queues[q]);
for (int i = 0; i < I; i++) router.setAgentGroup (i, agentGroups[i]);
router.setClearWaitingQueues (true);
router.addExitedContactListener (new MyContactMeasures());

mats.add (numArriv          = new ContactSumMatrix (K, P + 2));
mats.add (numBlocked        = new ContactSumMatrix (K, P + 2));
mats.add (numServed         = new ContactSumMatrix (K, P + 2));
mats.add (numGoodSL         = new ContactSumMatrix (K, P + 2));
mats.add (numAbandoned      = new ContactSumMatrix (K, P + 2));
mats.add (numDisconnected   = new ContactSumMatrix (K, P + 2));
mats.add (sumWaitingTimes   = new ContactSumMatrix (K, P + 2));

```

```

mats.add (sumServiceTimes = new ContactSumMatrix (K, P + 2));
for (int i = 0; i < I; i++)
    mats.add (vstat[i] = new NonStationaryMeasureMatrix<GroupVolumeStatMeasureMatrix>
        (pce, new GroupVolumeStatMeasureMatrix (agentGroups[i], K)));
svm = GroupVolumeStatMeasureMatrix.getServiceVolumeMeasureSet (vstat);
ivm = GroupVolumeStatMeasureMatrix.getIdleVolumeMeasureSet (vstat);
tvm = GroupVolumeStatMeasureMatrix.getTotalVolumeMeasureSet (vstat);
dsvm = GroupVolumeStatMeasureMatrix.getServiceVolumeMeasureSet (vstat, K);
dtvm = GroupVolumeStatMeasureMatrix.getTotalVolumeMeasureSet (vstat, K);

mTallies.add (arrivals = create
    ("Number of arrived contacts",
     TYPENAMES, PERIODNAMES));
mTallies.add (blocked = create
    ("Number of blocked contacts",
     TYPENAMES, PERIODNAMES));
mTallies.add (served = create
    ("Number of served contacts",
     TYPENAMES, PERIODNAMES));
serviceLevel = createRatio
    ("Service level",
     TYPENAMES, PERIODNAMES);
mTallies.add (occupancy = create
    ("Agents' occupancy ratio",
     GROUPNAMES, PERIODNAMES));
mTallies.add (occupancyPerType = create
    ("Occupancy ratio per contact type",
     GROUPTYPENAMES, PERIODNAMES));
mTallies.add (serviceTimes = create
    ("Service time",
     TYPENAMES, PERIODNAMES));
mTallies.add (busyCost = create
    ("Busy cost of agents",
     GROUPNAMES, PERIODNAMES));
mTallies.add (idleCost = create
    ("Idle cost of agents",
     GROUPNAMES, PERIODNAMES));
trunkAvail = new Tally
    ("Number of available trunks");
trunkBusy = new Tally
    ("Number of busy trunks");
mTallies.add (waitingTimes = create
    ("Average waiting time",
     TYPENAMES, PERIODNAMES));
mTallies.add (abandoned = create
    ("Number of abandoned contacts",
     TYPENAMES, PERIODNAMES));

```

```

    mTallies.add (disconnected = create
                  ("Number of disconnected contacts",
                   TYPENAMES, PERIODNAMES));
}

private MatrixOfTallies<Tally> create
(String name, String[] rowNames, String[] columnNames) {
    final MatrixOfTallies<Tally> mta = MatrixOfTallies.createWithTally
        (rowNames.length, columnNames.length);
    mta.setName (name);
    for (int r = 0; r < rowNames.length; r++)
        for (int c = 0; c < columnNames.length; c++)
            mta.get (r, c).setName (rowNames[r] + ", " + columnNames[c]);
    for (final Tally ta : mta) {
        ta.setConfidenceIntervalStudent();
        ta.setConfidenceLevel (LEVEL);
    }
    return mta;
}

private MatrixOfFunctionOfMultipleMeansTallies<FunctionOfMultipleMeansTally>
createRatio (String name, String[] rowNames, String[] columnNames) {
    final RatioFunction ratio = new RatioFunction();
    final MatrixOfFunctionOfMultipleMeansTallies<FunctionOfMultipleMeansTally> mta =
        MatrixOfFunctionOfMultipleMeansTallies.create
            (ratio, 2, rowNames.length, columnNames.length);
    mta.setName (name);
    for (int r = 0; r < rowNames.length; r++)
        for (int c = 0; c < columnNames.length; c++)
            mta.get (r, c).setName (rowNames[r] + ", " + columnNames[c]);
    for (final FunctionOfMultipleMeansTally ta : mta) {
        ta.setConfidenceIntervalDelta();
        ta.setConfidenceLevel (LEVEL);
    }
    return mta;
}

// Creates new contacts
class MyContactFactory implements ContactFactory {
    int type;
    MyContactFactory (int type) { this.type = type; }

    public Contact newInstance() {
        final Contact contact = new Contact (type);
        contact.setTrunkGroup (trunks);
        contact.setDefaultServiceTime (sgen[type].nextDouble());
        contact.setDefaultPatienceTime (pgen[type].nextDouble());
    }
}

```

```

        return contact;
    }
}

class MyRouter extends SingleFIFOQueueRouter {
    MyRouter (int[][] typeToGroupMap, int[][] groupToTypeMap) {
        super (typeToGroupMap, groupToTypeMap);
    }

    @Override
    protected EndServiceEvent selectAgent (Contact contact) {
        final int tid = contact.getTypeId();
        if (tid == BALANCE) {
            final AgentGroup group = AgentGroupSelectors.selectUniform
                (this, typeToGroupMap[tid], agentStream);
            if (group == null) return null;
            return group.serve (contact);
        }
        else
            return super.selectAgent (contact);
    }

    @Override
    protected DequeueEvent selectContact (AgentGroup group, Agent agent) {
        final int gid = group.getId();
        final int bestType = groupToTypeMap[gid][0];
        if (queues[bestType].isEmpty ())
            return super.selectContact (group, agent);
        else
            return queues[bestType].removeFirst (DEQUEUEUETYPE_BEGINSERVICE);
    }
}

class MyContactMeasures implements ExitedContactListener {
    public void blocked (Router router, Contact contact, int bType) {
        final int type = contact.getTypeId();
        final int period = pce.getPeriod (contact.getArrivalTime());
        numArriv.add (type, period, 1);
        numBlocked.add (type, period, 1);
    }

    public void dequeued (Router router, DequeueEvent ev) {
        final Contact contact = ev.getContact();
        final int type = contact.getTypeId();
        final int period = pce.getPeriod (contact.getArrivalTime());
        numArriv.add (type, period, 1);
        if (ev.getEffectiveDequeueType() == Router.DEQUEUEUETYPE_NOAGENT)

```

```

        numDisconnected.add (type, period, 1);
    else
        numAbandoned.add (type, period, 1);
    }

    public void served (Router router, EndServiceEvent ev) {
        final Contact contact = ev.getContact();
        final int type = contact.getTypeId();
        final int period = pce.getPeriod (contact.getArrivalTime());
        numArriv.add (type, period, 1);
        numServed.add (type, period, 1);
        final double qt = contact.getTotalQueueTime();
        if (qt < AWT[type]) numGoodSL.add (type, period, 1);
        final double st = contact.getTotalServiceTime();
        sumServiceTimes.add (type, period, st);
        sumWaitingTimes.add (type, period, contact.getTotalQueueTime());
    }
}

public void simulateOneDay() {
    Sim.init();
    pce.init();
    trunks.init();
    ContactCenter.initElements (arrivProc);
    ContactCenter.initElements (agentGroups);
    ContactCenter.initElements (queues);
    ContactCenter.initElements (mats);
    ContactCenter.toggleElements (arrivProc, true);
    pce.start();
    Sim.start();
    pce.stop();
    addObs();
}

void addObs() {
    final DoubleMatrix2D arrivalsM = RepSimCC.getReplicationValues (numArriv, false, false);
    arrivals.add (arrivalsM);
    final DoubleMatrix2D bl = RepSimCC.getReplicationValues (numBlocked, false, false);
    blocked.add (bl);
    final DoubleMatrix2D servedM = RepSimCC.getReplicationValues
        (numServed, false, false);
    served.add (servedM);
    final DoubleMatrix2D gslM = RepSimCC.getReplicationValues
        (numGoodSL, false, false);
    gslM.assign (Functions.mult (100));
    //addRatio (serviceLevel, gslM, arrivalsM);
    serviceLevel.addSameDimension (gslM, arrivalsM);
}

```

```

    final DoubleMatrix2D svM = RepSimCC.getReplicationValues
        (svm, false, false);
    final DoubleMatrix2D ivM = RepSimCC.getReplicationValues
        (ivm, false, false);
    final DoubleMatrix2D tvM = RepSimCC.getReplicationValues
        (tvm, false, false);
    StatUtil.addRatio (occupancy, svM, tvM, 100.0);
    final DoubleMatrix2D dsvM = RepSimCC.getReplicationValues
        (dsvm, false, false);
    final DoubleMatrix2D dtvM = RepSimCC.getReplicationValues
        (dtvm, false, false);
    StatUtil.addRatio (occupancyPerType, dsvM, dtvM, 100.0);

    MatrixUtil.getCost (svM, COST);
    MatrixUtil.getCost (ivM, COST);
    busyCost.add (svM);
    idleCost.add (ivM);
    final DoubleMatrix2D st = RepSimCC.getReplicationValues (sumServiceTimes, false, false);
    StatUtil.addRatio (serviceTimes, st, servedM);
    final DoubleMatrix2D wt = RepSimCC.getReplicationValues (sumWaitingTimes, false, false);
    StatUtil.addRatio (waitingTimes, wt, arrivalsM);
    final DoubleMatrix2D ab = RepSimCC.getReplicationValues (numAbandoned, false, false);
    abandoned.add (ab);
    final DoubleMatrix2D dis = RepSimCC.getReplicationValues (numDisconnected, false, false);
    disconnected.add (dis);
    trunkBusy.add (trunks.getStatLines().average());
    trunkAvail.add (trunks.getCapacity() - trunks.getStatLines().average());
}

public void simulate (int n) {
    ContactCenter.initElements (mTallies);
    serviceLevel.init();
    trunkAvail.init();
    trunkBusy.init();
    for (int r = 0; r < n; r++) simulateOneDay();
}

public void printStatistics() {
    System.out.println (trunkAvail.reportAndCIStudent (LEVEL, 3));
    System.out.println (trunkBusy.reportAndCIStudent (LEVEL, 3));

    System.out.println
        (arrivals.columnReport
         (arrivals.columns() - 1));
    System.out.println
        (blocked.columnReport
         (blocked.columns() - 1));
}

```



```

        System.out.println
            (serviceLevel.columnReport
             (serviceLevel.columns() - 1));
        System.out.println
            (serviceTimes.columnReport
             (serviceTimes.columns() - 1));
        System.out.println
            (waitingTimes.columnReport
             (waitingTimes.columns() - 1));
        System.out.println
            (abandoned.columnReport
             (abandoned.columns() - 1));
        System.out.println
            (disconnected.columnReport
             (disconnected.columns() - 1));
        System.out.println
            (occupancy.columnReport
             (occupancy.columns() - 1));
        System.out.println
            (idleCost.columnReport
             (idleCost.columns() - 1));
        System.out.println
            (busyCost.columnReport
             (busyCost.columns() - 1));
        System.out.println
            (occupancyPerType.columnReport
             (occupancyPerType.columns() - 1));
    }

    public static void main (String[] args) {
        final Bank b = new Bank();
        final Chrono timer = new Chrono();
        b.simulate (NUMDAYS);
        System.out.println ("CPU time: " + timer.format());
        b.printStatistics();
    }
}

```

The **Bank** class representing the simulator uses the **ContactCenter** utility class as in the previous example. Names are also associated with each contact type, agent group, period, and (contact type, agent group) pair, for clearer statistical reports.

This example demonstrates an alternative way for performing statistical collecting. Instead of built-in types or arrays for counting events, we use special *measure matrices* adapted for storing observations for multiple performance measures and periods. For example, `numArriv` is a $(K + 1) \times (P + 2)$ matrix whose rows correspond to contact types and columns

correspond to periods. The used contact sum matrix must be constructed with the number of contact types and periods, and provides an **add** method accepting an observation for contact type k during period p . If $K = 1$, the matrix contains a single row. Otherwise, each addition to element (k, p) is automatically repeated in the last row, at element (K, p) , to update the sum over all contact types. In this model, since the lengths of the preliminary and wrap-up periods are 0, the columns 0 and $P + 1$ of the matrix will always contain 0's. However, in general, these columns could contain non-zero values. Similar matrices are constructed for other statistics such as the number of blocked, abandoned, and served contacts.

For the occupancy ratio, as in the previous example, a statistical collector is constructed for each agent group. An agent group volume statistical collector is also a measure matrix, with rows for the number of busy agents, the number of working agents, etc. Since it contains a single column for the integrals over all simulation time, each matrix of agent group measures is wrapped into an integral measure matrix to get the integrals for multiple periods, by recording the values of integrals each time a period ends. For $N_{B,i}(t)$ and $N_i(t) + N_{G,i}(t)$, *measure sets* regrouping the values for each agent group during each period are constructed. A measure set is a matrix of measures whose rows are copied from other matrices. For example, **svm** is a measure set containing the service volumes: for $i = 0, \dots, I - 1$ and $p = 1, \dots, P$, element (i, p) corresponds to $\int_{t_{p-1}}^{t_p} N_{B,i}(t) dt$, and element (I, p) contains $\int_{t_{p-1}}^{t_p} N_B(t) dt$. Additionally, the program can compute $\int_{t_{p-1}}^{t_p} N_{B,i,k}(t) dt$, the number of busy agents in group i serving contacts of type k , because the **vstat** instances were constructed with K . To regroup the values, in **dsvm**, each row $j = Ki + k$, for $i = 0, \dots, I - 1$ and $k = 0, \dots, K - 1$, contains integrals for $N_{B,i,k}(t)$. For $i = I$, $KI + k$ contains the integrals for $N_{B,i}(t)$. **dtvm** is similar to **tvm**, except that each row is duplicated K times to get the same number of rows as **dsvm**.

For each matrix of measures, a matrix of collectors is declared. Each matrix of collectors contains the same number of rows than its corresponding matrix of measures, and $P + 1$ columns. The first P columns correspond to main periods while the last column contains time-aggregate values for all periods. Optionally, these aggregate values may include the preliminary and wrap-up periods, but this is not used in this example. Each matrix of probes has a global name corresponding to a performance measure, row names corresponding to type or group names, and column names corresponding to period names.

Matrices of measures and probes are created in the constructor to be added to lists. This allows them to be automatically initialized by the **initElements** method.

The **simulate** method initializes all probes in the list **repProbes** and calls **simulateOneDay** n times to perform the replications. Each replication initializes the simulator, the contact center components, and the matrices of measures. The arrival processes are turned on, and the simulation can start.

The **addObs** method uses a convenience method from **ContactCenter** and **RepSimCC** (another utility class) to convert matrices of measures into Colt matrices. Let M be a $R \times (P + 2)$ matrix of measures. For $r = 0, \dots, R - 1$ and $p = 1, \dots, P$, element $m(r, p)$ of the matrix of measures is copied into element $(r, p - 1)$ of the Colt matrix. Element (r, P)

is obtained by

$$\sum_{p=1}^P m(r, p),$$

for each $r = 0, \dots, R-1$. For computing the costs, another convenience method in **ContactCenter** is used.

When a contact is served by a specialist, the service time is multiplied by 0.75. However, at the time its service time is generated, if the contact is not served immediately, we do not know the group of its serving agent. This problem is addressed by changing the agent groups' service time generator. Similarly to waiting queues, a value generator is associated with an end-communication indicator. By default, service times are extracted from contacts and the end-communication type is 0. This can be changed to any **ValueGenerator** implementation by calling the **setContactTimeGenerator** method in **AgentGroup**. The default contact time generator supports multipliers similar to the talk time multipliers of Arena Contact Center Edition. To activate this feature, the program constructs a new **ContactServiceTimeGenerator** and assigns it to the end-communication type 0. For example, the Savings specialists will have a 0.75 service time multiplier for Savings contacts and 1.0 multipliers for other types. Note that this does not affect random number synchronization if it is used, since the random service times are generated at the contact's creation and multiplied by a constant afterward.

The agents' proficiency rewards preferred contacts-to-agents associations, but it is the task of the router to enforce the preference. The router is a modified **SingleFIFOQueueRouter** whose type-to-group map specifies that Savings contact types should be routed to Savings specialists first, then to Checking specialists if no Savings specialist is available. For Checking contact type, this is the reversed order. For Account Balance, any order may be used. The inherited routing policy implements the expected selection rule for the two first types, but it enforces an unwanted priority for Account Balance. We therefore need to override the **selectAgent** method to redefine the scheme for this particular contact type. The new method simply selects a Checking or Savings specialist randomly and uniformly among the available agents, or reverts to the superclass' selection rule if the contact type corresponds to Savings or Checking.

The contact selection rule of the single FIFO queue router is not appropriate for our needs, because it enforces no priority at all on waiting queues. We therefore need to override the **selectContact** method to add such priorities. The new method, which is called when an agent becomes free, first tries to pick a contact in the queue corresponding to the preferred type. The **removeFirst** method removes the first contact in queue with dequeue type **DEQUEUEUETYPE_BEGINSERVICE** (which corresponds to 0), and returns a dequeue event reference from which information can be extracted. Note that the argument to **removeFirst** is the dequeue type, not the index of the contact in the queue to be removed. The removed contact is extracted from the event and handled to the **serve** method of the agent group having a free member. If the prioritized queue is empty, the method falls back to the superclass' implementation which takes the contact with the longest waiting time.

Listing 26 gives the statistical results for the simulated Bank model. We can notice that the agents mainly serve Checking contacts. Also, if we sum up the per-contact type occupancy ratios for an agent group, we obtain its global occupancy ratio.

Listing 26: Results of the program Bank

```

CPU time: 0:0:3.48
REPORT on Tally stat. collector ==> Number of available trunks
  num. obs.      min      max      average      standard dev.
    1000      14.243      14.392      14.320      0.022
95.0% confidence interval for mean (student): (    14.318,    14.321 )

REPORT on Tally stat. collector ==> Number of busy trunks
  num. obs.      min      max      average      standard dev.
    1000      0.608      0.757      0.680      0.022
95.0% confidence interval for mean (student): (    0.679,    0.682 )

Report for Number of arrived contacts
                                num obs.      min      max      average      std. dev.      conf\
                                . int.
savings, all periods           1000      70.000      131.000      99.819      10.045      95.0%\
(  99.196, 100.442)
checking, all periods           1000      885.000      1097.000      1000.146      31.172      95.0%\
(  998.212, 1002.080)
account balance, all periods     1000      2349.000      2656.000      2502.340      50.017      95.0%\
( 2499.236, 2505.444)
all types, all periods           1000      3438.000      3787.000      3602.305      58.647

Report for Number of blocked contacts
                                num obs.      min      max      average      std. dev.      conf\
                                . int.
savings, all periods           1000      0.000      2.000      0.069      0.276      95.0%\
(   0.052,   0.086)
checking, all periods           1000      0.000      7.000      0.756      1.172      95.0%\
(   0.683,   0.829)
account balance, all periods     1000      0.000      15.000      1.826      2.245      95.0%\
(   1.687,   1.965)
all types, all periods           1000      0.000      22.000      2.651      3.125

Report for Service level
                                func. of averages      std. dev.      nobs.      conf. int.
savings, all periods           99.234      0.991      1000      95.0% (  99.172,  \
99.295)
checking, all periods           99.461      0.486      1000      95.0% (  99.431,  \
99.491)
account balance, all periods     97.301      1.032      1000      95.0% (  97.237,  \
97.365)

```

all types, all periods		97.954	0.813	1000		
Report for Service time						
	num obs.	min	max	average	std. dev.	conf\
	. int.					
savings, all periods	1000	2.664	5.059	3.781	0.383	95.0%\
(3.758, 3.805)						
checking, all periods	1000	3.383	4.269	3.852	0.127	95.0%\
(3.844, 3.860)						
account balance, all periods	1000	0.915	1.059	0.999	0.021	95.0%\
(0.998, 1.001)						
all types, all periods	1000	1.724	2.041	1.870	0.046	
Report for Average waiting time						
	num obs.	min	max	average	std. dev.	conf\
	. int.					
savings, all periods	1000	0.000	0.109	0.024	0.020	95.0%\
(0.023, 0.026)						
checking, all periods	1000	2.9E-3	0.106	0.028	0.014	95.0%\
(0.027, 0.029)						
account balance, all periods	1000	7.7E-3	0.086	0.036	0.014	95.0%\
(0.035, 0.037)						
all types, all periods	1000	7.7E-3	0.079	0.034	0.013	
Report for Number of abandoned contacts						
	num obs.	min	max	average	std. dev.	conf\
	. int.					
savings, all periods	1000	0.000	1.000	0.017	0.129	95.0%\
(9.0E-3, 0.025)						
checking, all periods	1000	0.000	4.000	0.161	0.501	95.0%\
(0.130, 0.192)						
account balance, all periods	1000	0.000	32.000	5.357	5.133	95.0%\
(5.038, 5.676)						
all types, all periods	1000	0.000	34.000	5.535	5.334	
Report for Number of disconnected contacts						
	num obs.	min	max	average	std. dev.	conf\
	. int.					
savings, all periods	1000	0.000	0.000	0.000	0.000	95.0%\
(0.000, 0.000)						
checking, all periods	1000	0.000	1.000	6.0E-3	0.077	95.0%\
(1.2E-3, 0.011)						
account balance, all periods	1000	0.000	3.000	0.017	0.157	95.0%\
(7.2E-3, 0.027)						
all types, all periods	1000	0.000	3.000	0.023	0.175	
Report for Agents' occupancy ratio						

	num obs.	min conf. int.	max	average	std. dev. \
savings specialists, all periods 95.0% (25.281, 25.425)	1000	21.425	29.202	25.353	1.160 \
checking specialists, all periods 95.0% (43.014, 43.172)	1000	39.273	47.129	43.093	1.270 \
all groups, all periods	1000	32.086	38.386	35.496	1.044
Report for Idle cost of agents					
	num obs.	min conf. int.	max	average	std. dev. \
savings specialists, all periods 95.0% (1209.121, 1211.453)	1000	1147.079	1273.639	1210.287	18.789 \
checking specialists, all periods 95.0% (768.951, 771.074)	1000	716.097	821.755	770.012	17.104 \
all groups, all periods	1000	1888.014	2078.247	1980.299	30.467
Report for Busy cost of agents					
	num obs.	min conf. int.	max	average	std. dev. \
savings specialists, all periods 95.0% (409.883, 412.219)	1000	347.282	473.466	411.051	18.827 \
checking specialists, all periods 95.0% (582.028, 584.174)	1000	531.028	638.323	583.101	17.290 \
all groups, all periods	1000	895.940	1083.427	994.152	30.607
Report for Occupancy ratio per contact type					
	num obs.	min conf. int.	max	average	std. dev\
savings spec./savings, all periods 95.0% (4.347, 4.425)	1000	2.726	6.887	4.386	0.630\
savings spec./checking, all periods 95.0% (4.986, 5.094)	1000	2.794	7.927	5.040	0.866\
savings spec./balance, all periods 95.0% (15.888, 15.966)	1000	14.192	18.409	15.927	0.624\
checking spec./savings, all periods 95.0% (0.191, 0.208)	1000	0.000	1.018	0.200	0.141\
checking spec./checking, all periods 95.0% (31.702, 31.863)	1000	27.526	36.326	31.783	1.300\
checking spec./balance, all periods 95.0% (11.083, 11.138)	1000	9.742	12.569	11.111	0.444\
all groups/savings, all periods 95.0% (1.975, 2.010)	1000	1.212	3.073	1.992	0.284\
all groups/checking, all periods 95.0% (20.273, 20.389)	1000	17.638	23.428	20.331	0.939\
all groups/balance, all periods	1000	11.818	14.360	13.173	0.386

7 Teamwork contact center

This models a contact center with complex routing logic in which a contact is processed by several agents. This model contains a single contact type but several agent groups. Contacts arrive following a non-homogeneous Poisson process with a mean arrival rate of 200/h from 8AM to 9AM and 100/h from 9AM to 5PM, and are served by several agents sequentially.

When a contact enters the system, it is routed to one of the two receptionists, or queued if the receptionists are not available. Service times are i.i.d. variables following the triangular distribution with minimum 0.5 minutes, mode 1 minute, and maximum 5 minutes. After the service is over, with probability 0.2, a *conference* is initiated with the accounting department if the corresponding agent is free. When a conference occurs, the contact still keeps the receptionist busy and requests a communication with a secondary agent, from the accounting agent group, if available. The service time with the accounting agent is uniform $2 * U(0, 1)$ minutes. The time spent with the receptionist is the sum of the triangular service time, and the uniform conference time if conferencing was required and succeeded. After the communication is over, the receptionist transfers the contact to the manager or to the technical support. Before going idle or accepting a new contact, the agent performs some after-contact work, e.g., updating a folder. The after-contact work has an exponential duration with mean 1 minute, and starts after the contact is transferred, not after it leaves the contact center. After-contact work is also performed by the technical support and manager with the same distribution, but it is not performed by conferenced agents.

With probability 0.05, the contact is transferred to the manager by the receptionist before after-contact work begins. If the manager is unavailable, the contact can be queued. After being served by the manager, the contact exits the system. The service time with the manager is generated from the same distribution as the service time with the receptionist, but the generated value is multiplied by 3.

With probability 0.95, the receptionist transfers the contact it has served to one of the five technical support agents instead of the manager. In contrast with the manger, if the contact cannot be served immediately, it disconnects without waiting in queue. The service time with the technical support is generated from the same triangular distribution as the receptionist, but the obtained value is multiplied by 10. After the service is done, with probability 0.2, a conference with the development agent is initiated. If the developer is available, the conference time is uniform $5 * U(0, 1)$.

Listing 27: A simulation program for a teamwork model

```
import umontreal.iro.lecuyer.contactcenters.ContactCenter;
import umontreal.iro.lecuyer.contactcenters.PeriodChangeEvent;
import umontreal.iro.lecuyer.contactcenters.ValueGenerator;
import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.contact.ContactFactory;
import umontreal.iro.lecuyer.contactcenters.contact.PiecewiseConstantPoissonArrivalProcess;
import umontreal.iro.lecuyer.contactcenters.contact.TrunkGroup;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
```

```

import umontreal.iro.lecuyer.contactcenters.queue.StandardWaitingQueue;
import umontreal.iro.lecuyer.contactcenters.queue.WaitingQueue;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.server.Agent;
import umontreal.iro.lecuyer.contactcenters.server.AgentGroup;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;
import umontreal.iro.lecuyer.contactcenters.server.GroupVolumeStat;
import umontreal.iro.lecuyer.probdist.TriangularDist;
import umontreal.iro.lecuyer.probdist.UniformDist;
import umontreal.iro.lecuyer.randvar.ExponentialGen;
import umontreal.iro.lecuyer.randvar.RandomVariateGen;
import umontreal.iro.lecuyer.randvar.TriangularGen;
import umontreal.iro.lecuyer.randvar.UniformGen;
import umontreal.iro.lecuyer.rng.MRG32k3a;
import umontreal.iro.lecuyer.rng.RandomStream;
import umontreal.iro.lecuyer.simevents.Event;
import umontreal.iro.lecuyer.simevents.Sim;
import umontreal.iro.lecuyer.stat.FunctionOfMultipleMeansTally;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.stat.list.ListOfTallies;
import umontreal.iro.lecuyer.util.Chrono;
import umontreal.iro.lecuyer.util.RatioFunction;

public class Teamwork {
    // Agent group identifiers
    static final int RECEPTION      = 0;
    static final int ACCOUNTING     = 1;
    static final int TECHSUP        = 2;
    static final int DEV            = 3;
    static final int MANAGER        = 4;
    static final String[] GROUPNAMES = {
        "Reception", "Accounting", "Technical support", "Development",
        "Manager", "All groups"
    };

    // Input data (times are in minutes)
    static final int P              = 24*7;
    static final double PERIODDURATION = 60;
    static final int NUMQUEUES      = 2;
    static final int I              = 5;
    static final int STARTHOUR      = 8;
    static final int ENDETHOUR      = 17;
    static final int NUMTRUNKS      = 25;
    static final double FHARRIVALRATE = 200;
    static final double ARRIVALRATE = 100;
    static final double PATIENCETIME = 2.0;

```



```

static final double MINSERVICETIME = 0.5;
static final double MODESERVICETIME = 1.0;
static final double MAXSERVICETIME = 5.0;
// Mean time for after contact work
static final double ACWTIME = 1.0;
static final double PROBCONFACCOUNTING = 0.2;
static final double PROBTRANSMANAGER = 0.05;
static final double PROBCONFDEV = 0.2;
// Acceptable waiting time
static final double AWT = 120/60.0;
// 2 receptionists, 5 technical support, and 1 agent in each other groups
static final int[] NUMAGENTS = { 2, 1, 5, 1, 1 };
// Service time is multiplied by 10 for technical support,
// and by 3 for the manager.
// Conference time is multiplied by 2 for accounting,
// and by 5 for developer.
static final double[] STIMEMULT = { 1, 2, 10, 3, 5 };
static final double LEVEL = 0.95; // Level of conf. int.
static final int NUMDAYS = 1000;

// Known expectations
static double EXPARRIVALS = (FHARRIVALRATE +
                             (ENDHOUR - STARTHOUR - 1)*ARRIVALRATE)*7;
static double[] EXPNUMAGENTS = new double[I + 1];
static {
    for (int i = 0; i < I; i++) {
        EXPNUMAGENTS[i] = 7*PERIODDURATION*(ENDHOUR - STARTHOUR)*NUMAGENTS[i];
        EXPNUMAGENTS[I] += EXPNUMAGENTS[i];
    }
}

// Contact center components
PeriodChangeEvent pce; // Event marking new periods
TrunkGroup trunks; // Manages phone lines
PiecewiseConstantPoissonArrivalProcess customers;
AgentGroup[] groups = new AgentGroup[I];
WaitingQueue[] queues = new WaitingQueue[NUMQUEUES];
Router router;

// Random number generators
RandomVariateGen pgen = new ExponentialGen (new MRG32k3a(), 1.0/PATIENCETIME);
RandomVariateGen sgen = new TriangularGen
    (new MRG32k3a(), new TriangularDist
     (MINSERVICETIME, MAXSERVICETIME, MODESERVICETIME));
RandomVariateGen acwgen = new ExponentialGen (new MRG32k3a(), 1.0/ACWTIME);
RandomVariateGen confGen = new UniformGen (new MRG32k3a(), 0, 1);
RandomStream probConfAccountingStream = new MRG32k3a();

```

```

RandomStream probTransManagerStream = new MRG32k3a();
RandomStream probConfDevStream = new MRG32k3a();

// Counters used during replications
int numArrived, numOffered, numBlocked, numAbandoned,
    numGoodSL, numServed, numDisconnected;
double[] Nb = new double[I + 1];
double sumWaitingTimes, sumServiceTimes, sumTotalTimes;
GroupVolumeStat[] vstat = new GroupVolumeStat[I];

// Statistical probes
Tally arrived = new Tally ("Number of arrived contacts");
Tally offered = new Tally ("Offered load");
Tally served = new Tally ("Number of served contacts");
Tally disconnected = new Tally ("Number of disconnected contacts");
Tally goodSL = new Tally ("Number of contacts in target");
Tally serviceLevel = new Tally ("Service level");
Tally blocked = new Tally ("Number of blocked contacts");
Tally abandoned = new Tally ("Number of abandoned contacts");
Tally totalTime = new Tally ("Sojourn time of contacts");
FunctionOfMultipleMeansTally speedOfAnswer =
    new FunctionOfMultipleMeansTally (new RatioFunction(), "Speed of answer", 2);
Tally serviceTime = new Tally ("Service time");
ListOfTallies<Tally> occupancy = ListOfTallies.createWithTally (I + 1);
Tally trunkBusy = new Tally ("Number of busy trunks");
Tally trunkAvail = new Tally ("Number of available trunks");

Teamwork() {
    occupancy.setName ("Agents' occupancy ratio");
    for (int i = 0; i <= I; i++) {
        occupancy.get (i).setName (GROUPNAMES[i]);
        occupancy.get (i).setConfidenceIntervalStudent();
        occupancy.get (i).setConfidenceLevel (LEVEL);
    }
    pce = new PeriodChangeEvent (PERIODDURATION, P + 2, 0.0);
    trunks = new TrunkGroup (NUMTRUNKS);
    trunks.setStatCollecting (true);

    final double[] arrivalRates = new double[P + 2];
    for (int day = 0; day < 7; day++) {
        arrivalRates[24*day + STARTHOUR + 1] = FHARRIVALRATE;
        for (int hour = STARTHOUR + 1; hour < ENDDHOUR; hour++)
            arrivalRates[24*day + hour + 1] = ARRIVALRATE;
    }
    customers = new PiecewiseConstantPoissonArrivalProcess
        (pce, new MyContactFactory(), arrivalRates, new MRG32k3a());
    customers.setNormalizing (true);
}

```

```

final int[][] numAgents = new int[I][P + 2];
for (int day = 0; day < 7; day++)
    for (int hour = STARTHOUR; hour < ENDFHOUR; hour++) {
        final int ind = 24*day + hour + 1;
        for (int i = 0; i < I; i++)
            numAgents[i][ind] = NUMAGENTS[i];
    }
for (int i = 0; i < I; i++) {
    groups[i] = new AgentGroup (pce, numAgents[i]);
    groups[i].setAfterContactTimeGenerator
        (0, new AfterContactGen (i));
    vstat[i] = new GroupVolumeStat (groups[i]);
}
for (int q = 0; q < queues.length; q++)
    queues[q] = new StandardWaitingQueue();
queues[1].setMaximalQueueTimeGenerator
    (1, new ValueGenerator() {
        public void init() {}

        public double nextDouble (Contact contact) {
            return ((MyContact)contact).pManager;
        }
    });

router = new MyRouter();
router.setClearWaitingQueues (true);
customers.addNewContactListener (router);
for (int q = 0; q < queues.length; q++)
    router.setWaitingQueue (q, queues[q]);
for (int i = 0; i < I; i++) router.setAgentGroup (i, groups[i]);
router.addExitedContactListener (new MyContactMeasures());
}

// Create new contacts
class MyContactFactory implements ContactFactory {
    public Contact newInstance() {
        final Contact contact = new MyContact();
        contact.setTrunkGroup (trunks);
        return contact;
    }
}

// Contact subclass with custom attributes
class MyContact extends Contact {
    double pManager;
    double[] stime = new double[I];

```

```

double[] acwTime = new double[I];
double uConfAccounting;
double uTransManager;
double uConfDev;

MyContact() {
    setDefaultPatienceTime (pgen.nextDouble());
    pManager = pgen.nextDouble();
    for (int i = 0; i < I; i++) {
        double baseServiceTime;
        if (i == ACCOUNTING || i == DEV) {
            baseServiceTime = confGen.nextDouble();
            acwTime[i] = 0;
        }
        else {
            baseServiceTime = sgen.nextDouble();
            acwTime[i] = acwgen.nextDouble();
        }
        stime[i] = STIMEMULT[i]*baseServiceTime;
    }

    uConfAccounting = probConfAccountingStream.nextDouble();
    uTransManager = probTransManagerStream.nextDouble();
    uConfDev = probConfDevStream.nextDouble();
}

}

// Custom router
class MyRouter extends Router {
    MyRouter() {
        super (1, NUMQUEUES, I);
    }

    @Override
    public boolean canServe (int i, int k) {
        return true;
    }

    // Agent selection for incoming contacts
    @Override
    protected EndServiceEvent selectAgent (Contact contact) {
        if (groups[RECEPTION].getNumFreeAgents() == 0)
            return null;
        final EndServiceEvent es = groups[RECEPTION].serve (contact);
        return es;
    }
}

```

```
// Queue selection for incoming contacts, if agent selection fails
@Override
protected DequeueEvent selectWaitingQueue (Contact contact) {
    return queues[0].add (contact);
}

private WaitingQueue getQueue (int gid) {
    WaitingQueue queue = null;
    switch (gid) {
        case RECEPTION:
            queue = getWaitingQueue (0);
            break;
        case MANAGER:
            queue = getWaitingQueue (1);
            break;
    }
    return queue;
}

// Contact selection for receptionists and manager
@Override
protected DequeueEvent selectContact (AgentGroup group, Agent agent) {
    final int gid = group.getId();
    final WaitingQueue queue = getQueue (gid);
    if (queue == null)
        return null;
    if (queue.size() == 0)
        return null;
    return queue.removeFirst (DEQUEUEUETYPE_BEGINSERVICE);
}

// Clear queues when agents are off-duty
@Override
protected void checkWaitingQueues (AgentGroup group) {
    final int gid = group.getId();
    final WaitingQueue queue = getQueue (gid);
    if (queue == null)
        return;
    if (!mustClearWaitingQueue (queue.getId()))
        return;
    if (group.getNumAgents() == 0)
        queue.clear (DEQUEUEUETYPE_NOAGENT);
}

// Determines the service times, and manages conferencing
@Override
protected void beginService (EndServiceEvent es) {
```

```

final int gid = es.getAgentGroup().getId();
final MyContact contact = (MyContact)es.getContact();
switch (gid) {
case RECEPTION:
    if (contact.uConfAccounting < PROBCONFACCOUNTING)
        new ConferenceEvent (es, groups[ACCOUNTING]).schedule
            (contact.stime[RECEPTION]);
    else
        es.schedule (contact.stime[RECEPTION]);
    break;
case TECHSUP:
    if (contact.uConfDev < PROBCONFDEV)
        new ConferenceEvent (es, groups[DEV]).schedule
            (contact.stime[TECHSUP]);
    else
        es.schedule (contact.stime[TECHSUP]);
    break;
case ACCOUNTING:
case DEV:
    break;
default:
    es.schedule (contact.stime[gid]);
}
}

// At the end of the communication with an agent,
// manages transfer or exit
@Override
protected void endContact (EndServiceEvent es) {
    final AgentGroup group = es.getAgentGroup();
    final MyContact contact = (MyContact)es.getContact();
    if (group == groups[ACCOUNTING] ||
        group == groups[DEV])
        return;
    else if (group == groups[RECEPTION]) {
        if (contact.uTransManager < PROBTRANSMANAGER) {
            // Transfer to manager
            if (groups[MANAGER].getNumAgents() == 0)
                // Disconnected contact, but counted in offered load
                exitBlocked (contact, 5);
            else if (groups[MANAGER].getNumFreeAgents() == 0)
                queues[1].add (contact);
            else
                groups[MANAGER].serve (contact);
        }
        else // Transfer to technical support
            if (groups[TECHSUP].getNumFreeAgents() == 0)

```

```

        // Disconnected contact counted in offered load
        exitBlocked (contact, 5);
    else
        groups[TECHSUP].serve (contact);
    }
    else
        // The contact exits at the manager or technical support
        exitServed (es);
}

// Event happening at the end of a service, to manage conferencing
class ConferenceEvent extends Event {
    // Secondary agent group to conference with (accounting or developer)
    AgentGroup targetGroup;
    // End-service event for primary agent
    EndServiceEvent es;
    // End-service event for secondary agent
    EndServiceEvent esConf;

    ConferenceEvent (EndServiceEvent es,
                    AgentGroup targetGroup) {
        this.es = es;
        this.targetGroup = targetGroup;
    }

    @Override
    public void actions() {
        final MyContact contact = (MyContact)es.getContact();
        if (esConf != null ||
            targetGroup.getNumFreeAgents() == 0) {
            // End conferencing or abort it because
            // there is no agent in the target group.
            if (esConf != null)
                esConf.getAgentGroup().endContact (esConf, 0);
            es.getAgentGroup().endContact (es, 0);
        }
        else {
            // Start the conference: the contact is
            // served simultaneously by two agents.
            esConf = targetGroup.serve (contact);
            schedule (contact.stime[targetGroup.getId()]);
        }
    }
}

// Extracts after-contact work from contacts

```

```
class AfterContactGen implements ValueGenerator {
    private int groupId;

    AfterContactGen (int groupId) {
        this.groupId = groupId;
    }

    public void init() {}

    public double nextDouble (Contact contact) {
        return ((MyContact)contact).acwTime[groupId];
    }
}

// Updates counters
class MyContactMeasures implements ExitedContactListener {
    public void blocked (Router router, Contact contact, int bType) {
        ++numArrived;
        if (bType == 5) {
            ++numOffered;
            ++numDisconnected;
        }
        else
            ++numBlocked;
    }

    public void dequeued (Router router, DequeueEvent ev) {
        ++numArrived;
        ++numOffered;
        if (ev.getEffectiveDequeueType() == Router.DEQUEUE_TYPE_NOAGENT)
            ++numDisconnected;
        else
            ++numAbandoned;
    }

    public void served (Router router, EndServiceEvent ev) {
        final Contact contact = ev.getContact();
        ++numArrived;
        ++numOffered;
        final double qt = contact.getTotalQueueTime();
        if (qt < AWT) ++numGoodSL;
        sumWaitingTimes += qt;
        final double st = ev.getEffectiveContactTime();
        sumServiceTimes += st;
        ++numServed;
        sumTotalTimes += contact.getTotalQueueTime()
            + contact.getTotalServiceTime();
    }
}
```



```

    }
}

void simulateOneDay() {
    Sim.init();
    // Initializes period-change event, trunk group, arrival process,
    // waiting queues, and agent groups.
    pce.init();
    trunks.init();
    customers.init();
    ContactCenter.initElements (groups);
    ContactCenter.initElements (queues);
    numArrived = numOffered = numBlocked = numAbandoned = 0;
    sumWaitingTimes = sumServiceTimes = sumTotalTimes = 0;
    numGoodSL = numServed = numDisconnected = 0;
    for (int i = 0; i < I; i++) vstat[i].init();
    customers.start();
    pce.start();
    Sim.start();
    pce.stop();
    addObs();
}

void addObs() {
    arrived.add (numArrived);
    offered.add (numOffered);
    served.add (numServed);
    disconnected.add (numDisconnected);
    goodSL.add (numGoodSL);
    blocked.add (numBlocked);
    abandoned.add (numAbandoned);
    serviceTime.add (sumServiceTimes/numServed);
    totalTime.add (sumTotalTimes/numServed);
    serviceLevel.add (100.0*numGoodSL/EXPARRIVALS);
    speedOfAnswer.add (sumWaitingTimes, numServed);

    trunkBusy.add (trunks.getStatLines().average());
    trunkAvail.add (trunks.getCapacity() - trunks.getStatLines().average());

    Nb[I] = 0;
    for (int i = 0; i < I; i++) {
        Nb[i] = vstat[i].getStatNumBusyAgents().sum();
        Nb[I] += Nb[i];
        Nb[i] /= EXPNUMAGENTS[i]/100;
    }
    Nb[I] /= EXPNUMAGENTS[I]/100;
    occupancy.add (Nb);
}

```

```
}

void simulate (int n) {
    arrived.init();
    offered.init();
    served.init();
    disconnected.init();
    goodSL.init();
    serviceLevel.init();
    blocked.init();
    abandoned.init();
    totalTime.init();
    speedOfAnswer.init();
    serviceTime.init();
    occupancy.init();
    trunkBusy.init();
    trunkAvail.init();
    for (int r = 0; r < n; r++) simulateOneDay();
}

public void printStatistics() {
    System.out.println (trunkAvail.reportAndCISudent (LEVEL, 3));
    System.out.println (trunkBusy.reportAndCISudent (LEVEL, 3));
    System.out.println (arrived.reportAndCISudent (LEVEL, 3));
    System.out.println (offered.reportAndCISudent (LEVEL, 3));
    System.out.println (served.reportAndCISudent (LEVEL, 3));
    System.out.println (disconnected.reportAndCISudent (LEVEL, 3));
    System.out.println (blocked.reportAndCISudent (LEVEL, 3));
    System.out.println (goodSL.reportAndCISudent (LEVEL, 3));
    System.out.println (abandoned.reportAndCISudent (LEVEL, 3));
    System.out.println (totalTime.reportAndCISudent (LEVEL, 3));
    System.out.println (speedOfAnswer.reportAndCIDelta (LEVEL, 3));
    System.out.println (serviceTime.reportAndCISudent (LEVEL, 3));
    System.out.println (serviceLevel.reportAndCISudent (LEVEL, 3));
    System.out.println (occupancy.report ());
}

public static void main (String[] args) {
    final Teamwork t = new Teamwork();
    final Chrono timer = new Chrono();
    t.simulate (NUMDAYS);
    System.out.println ("CPU time: " + timer.format());
    t.printStatistics();
}
}
```

Listing 27 presents the complete source code for this example. In the program, a class `Teamwork` is created, and constants are declared for input data. Most events are counted using integers since the system supports a single contact type. The occupancy ratios of agents require arrays because the system has five agent groups.

In previous examples, the different steps of a service were not modeled explicitly. An exponential or uniform service time was generated and used, and the end of service was automatically scheduled by `ContactCenters`. In this example, the service of a contact is divided in three parts: the communication time with the agent, the optional conference time with another agent, and the after-contact time. The library supports after-contact time, but conferencing must be implemented by the user, because an infinity of models can be imagined.

In `ContactCenters`, the service of a contact is a two-pass process composed of the communication with the agent (first phase), and the after-contact work (second phase). When the first phase is over, the contact is notified to exited-contact listeners by the router for statistical purposes, but the busy agent is not freed up until the second phase is over. In previous examples, since there was no after-contact work, the two steps were ending at the same time.

The `setDefaultServiceTime` method of `Contact` we used in previous examples sets the contact time to the service time, and resets the after-contact time to 0. The two durations can be changed separately by using the `setDefaultContactTime` and `setDefaultAfterContactTime` methods, respectively.

In section 4.3, we used an agent-group listener to track the number of agents in a group, in order to disable statistical collecting when $N(t) = 0$. Because such a listener can be notified about the end of a service, one could think about using this mechanism to implement conferencing. However, conferencing cannot be modeled with an agent-group listener because the listener is notified only after the phases of the service are terminated. It is not possible to conditionally extend the service duration as it is needed in a conferencing model. To model such complex services, automatic service termination must be disabled. By default, contact times are extracted from contact objects and are infinite if unspecified. As with the waiting queue which does not schedule the dequeue event when the maximal queue time is infinite, the end-service event is not scheduled if the obtained service time is infinite. The event can then be manually scheduled to simulate a simple service, or a wrapper event can be used to add logic before the service termination.

Usually, a patience, a contact (or handle) and an after-contact times are the only random variables associated with a contact. Here, additional variables are required: the contact time for each agent group (including conference times), the after-contact times, as well as probabilistic branching decision variables. These variates can be generated when needed, but to maximize random number synchronization, it is better to generate all random variates at the creation of the contact object since all the information is already available at this moment. We do not know in advance the exact path of the contact, but we know which random variates it might need during its lifecycle. To support this extra information, a

subclass of `Contact` called `MyContact` is created. The `MyContactFactory` instantiates `MyContact` objects instead of `Contact` objects as before. Unfortunately, the library's classes manage `Contact` instances which must be casted to `MyContact` as needed.

For each agent group not used for conferencing, i.e., reception, manager and technical support, an exponential after-contact time is generated and stored in contacts. A custom value generator is assigned to each agent group in order to extract the appropriate time from the contact objects. The after-contact time generation works the same way as the service time and maximal queue time generation.

The router is the most complex part of this contact center. This time, we define a completely custom router which extends the `Router` abstract class. The abstract router is constructed with the supported number of contact types, agent groups, and waiting queues. In this example, there are two waiting queues: a first one for receptionists, and a second one for the manager. There are five agent groups: Reception, Accounting, Manager, Technical support, and Development. Every incoming contact is sent to the Reception by `selectAgent`. If there is no free receptionist, `selectAgent` returns `null` and the contact is queued by `selectWaitingQueue`.

When the communication part of a service is terminated, the appropriate agent group notifies it to the router which calls its protected method `endContact`. By default, this method simply calls `exitServed` in order for the contact to leave the system after service. We can implement contact transfer by overriding this method as follows. The end-contact event is ignored for the Accounting and Development agent groups since the method will be called again to end the communication with the associated Reception or Technical support agent groups. When the communication between the contactor and the receptionist is terminated, the transfer occurs: with probability 0.05, the contact is sent to the manager. If the contact center is closed at transfer time, the contactor is disconnected by calling the `exitBlocked` method. The blocking-type indicator, given to `exitBlocked`, is set to 5 in order to distinguish this disconnection from true blocking because of no phone line. If there is no free manager, the contact is put in the queue for the manager, i.e., the second waiting queue. Otherwise, the second service of the contact begins. If the contact is transferred to technical support, the system applies the same algorithm as for the manager, except that there is no waiting queue and a conference can occur. If no technical support agent is available, the service of the contact ends at the receptionist (`exitBlocked` is called). For any other agent group, the `endContact` method calls `exitServed` for the served contact to leave the system.

When the service of a contact by an agent begins, whether it is started by agent or contact selection, the protected `beginService` method is called. We override the method in our custom router to schedule the end-service event appropriately, or to construct and schedule a wrapper event for conferencing. For the case of the receptionist, with probability 0.2, a wrapper `ConferenceEvent` is scheduled to manage the service termination. The end-service event is kept inside the conference event for future use. With probability 0.8, the regular end-service event is scheduled and no conference event is constructed. Similar logic is used for the technical support, with a different secondary agent group. In the case of the manager, the regular end-service event is always scheduled. For Accounting and Development, nothing happens in that method, because the service termination is managed by the conferencing event.

Conferencing is implemented using an auxiliary wrapper event containing three fields: **es** for the end-service event associated with the main agent, **esConf** for the end-service event bound to the conferenced agent, and **targetGroup**, a reference to the conferenced agent group. The end-service event objects are only used as information containers instead of being scheduled as simulation events. The conference event happens at most twice: one time to start the conference, and a second time to terminate the communication with the main agent as well as the conference. The conference happens only if the conferenced agent is free.

Automatic waiting queue clearing must be redefined since there is no default data structure in use. The **checkWaitingQueues** method, which is called each time $N_i(t) = 0$ for an agent group i , is responsible for this operation. If it is called for a Reception agent, the Reception waiting queue is cleared. If it is called for the Manager, the Manager waiting queue is cleared. Otherwise, nothing is done. The waiting queue is cleared only if there is no available agent in the corresponding group.

The average speed of answer and service level performance measures can be computed several ways. Each contact in the system can sojourn in the reception's waiting queue, the manager's waiting queue, or both queues. Using the **getTotalQueueTime** method of the class **Contact** as in the previous examples gives the cumulative waiting time in both queues. We could also get the queue time for each waiting queue. Here, to keep the program as simple as possible, the speed of answer is assumed to be the total waiting time in both queues for served contacts. The average handle time is computed by taking the contact time for the last agent, which is extracted from the end-service event given to the **served** method of **MyContactMeasures**.

There are two types of disconnections: when agents go off-duty or when no technical support agent is free. The first case is handled in **dequeued** as usual. The second case is handled in **blocked** since the disconnection was performed using **exitBlocked**, in the **endContact** method of **MyRouter**. The blocking type is tested to distinguish this event from a blocking due to no available phone line.

We can see from the statistical report on Listing 28 that our results are different from Arena Contact Center Edition 8.0's. This is due to several factors and possible bugs in Arena Contact Center Edition arising when contacts are served by several agents sequentially.

First, there are several ways to define the same performance measure and Rockwell's documentation often does not clearly specify what is estimated when service is performed by multiple agents. By experimenting with simplified models, it is sometimes but not always possible to guess what computation is made. Inconsistencies such as service levels greater than 100% even arise in some extreme conditions. Without accessing the definitions of the Contact Center Arena templates, which cannot be done without the Professional edition, an explanation for these inconsistencies cannot be found.

Listing 28: Results of the program **Teamwork**

```
CPU time: 0:0:10.5
REPORT on Tally stat. collector ==> Number of available trunks
```

num. obs.	min	max	average	standard dev.
1000	21.665	21.813	21.739	0.025
95.0% confidence interval for mean (student): (21.738, 21.741)				

REPORT on Tally stat. collector ==> Number of busy trunks

num. obs.	min	max	average	standard dev.
1000	3.187	3.335	3.261	0.025
95.0% confidence interval for mean (student): (3.259, 3.262)				

REPORT on Tally stat. collector ==> Number of arrived contacts

num. obs.	min	max	average	standard dev.
1000	6766.000	7281.000	7003.654	86.815
95.0% confidence interval for mean (student): (6998.267, 7009.041)				

REPORT on Tally stat. collector ==> Offered load

num. obs.	min	max	average	standard dev.
1000	6766.000	7281.000	7003.628	86.816
95.0% confidence interval for mean (student): (6998.241, 7009.015)				

REPORT on Tally stat. collector ==> Number of served contacts

num. obs.	min	max	average	standard dev.
1000	828.000	920.000	866.968	13.219
95.0% confidence interval for mean (student): (866.148, 867.788)				

REPORT on Tally stat. collector ==> Number of disconnected contacts

num. obs.	min	max	average	standard dev.
1000	1264.000	1419.000	1339.118	23.725
95.0% confidence interval for mean (student): (1337.646, 1340.590)				

REPORT on Tally stat. collector ==> Number of blocked contacts

num. obs.	min	max	average	standard dev.
1000	0.000	5.000	0.026	0.260
95.0% confidence interval for mean (student): (9.9E-3, 0.042)				

REPORT on Tally stat. collector ==> Number of contacts in target

num. obs.	min	max	average	standard dev.
1000	457.000	593.000	522.891	18.231
95.0% confidence interval for mean (student): (521.760, 524.022)				

REPORT on Tally stat. collector ==> Number of abandoned contacts

num. obs.	min	max	average	standard dev.
1000	4555.000	5089.000	4797.542	88.275
95.0% confidence interval for mean (student): (4792.064, 4803.020)				

REPORT on Tally stat. collector ==> Sojourn time of contacts

num. obs.	min	max	average	standard dev.
1000	23.917	26.069	25.069	0.333

95.0% confidence interval for mean (student): (25.049, 25.090)

REPORT on Tally stat. collector ==> Speed of answer

func. of averages standard dev. num. obs.

1.774 0.056 1000

95.0% confidence interval for function of means: (1.771, 1.778)

REPORT on Tally stat. collector ==> Service time

num. obs. min max average standard dev.

1000 19.818 21.825 20.828 0.326

95.0% confidence interval for mean (student): (20.808, 20.848)

REPORT on Tally stat. collector ==> Service level

num. obs. min max average standard dev.

1000 6.529 8.471 7.470 0.260

95.0% confidence interval for mean (student): (7.454, 7.486)

Report for Agents' occupancy ratio

	num obs.	min	max	average	std. dev.	conf. \
	int.					

Reception	1000	97.413	98.917	98.258	0.241	95.0% (\
				98.243,	98.273)	

Accounting	1000	9.607	12.471	10.995	0.514	95.0% (\
				10.963,	11.027)	

Technical support	1000	93.262	96.251	94.657	0.416	95.0% (\
				94.631,	94.682)	

Development	1000	4.102	7.293	5.663	0.487	95.0% (\
				5.632,	5.693)	

Manager	1000	20.453	35.089	27.244	2.463	95.0% (\
				27.092,	27.397)	

All groups	1000	70.492	72.448	71.370	0.320	95.0% (\
				71.350,	71.390)	

8 Blend call center

This example is a simplified version of the model analyzed in [1], with reduced choice of distributions and dialing policies. Although it is a special case of the blend/multi-skill generic simulator, it demonstrates how to use a dialer as well as per-period probability distributions. The simulated day starts at 8AM, ends at 2PM, and is divided into three two-hours periods. Calls received by the center are denoted *inbound calls* and arrive following a Poisson process with randomized arrival rates $B\lambda_p$. To keep the agents busy, a dialer is used to perform *outbound calls* at some times during the day. Only a portion of the agents, denoted *blend*, are qualified for handling these calls in addition to inbound ones.

Before 11AM, there is no outbound call, so all agents perform inbound calls only. After 11AM, a portion of the agents becomes blend, and the dialer starts. Inbound agents still serve inbound calls only while calls that cannot be served immediately overflow to blend agents. From 11AM to five minutes before the end of the day, each time a service ends, the dialer tries to compose some phone numbers. The probability of a caller to be reached, or equivalently of *right party connect*, is specified for each period. Reached customers produce outbound calls which can only be served by blend agents. Outbound calls then follow a non-Poisson arrival process which cannot be modeled in simulation software such as Arena Contact Center Edition.

If no blend agent is available when an outbound call succeeds, the call is put in queue and considered as a *mismatch*. The mismatched calls are put into a different waiting queue than the inbound calls in order to be served with the greatest priority, because most reached callers will hang up without even waiting. Since in this model, there is no delay between the dialing and the right party connect, mismatches only occur when the dialer tries to call more customers than the number of free blend agents.

The program, shown in Listing 29, is very similar to the other examples, except that a dialer is used.

Listing 29: A blend call center model

```
import umontreal.iro.lecuyer.contactcenters.ConstantValueGenerator;
import umontreal.iro.lecuyer.contactcenters.PeriodChangeEvent;
import umontreal.iro.lecuyer.contactcenters.PeriodChangeListener;
import umontreal.iro.lecuyer.contactcenters.ToggleEvent;
import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.contact.ContactFactory;
import umontreal.iro.lecuyer.contactcenters.contact.NewContactListener;
import umontreal.iro.lecuyer.contactcenters.contact.PiecewiseConstantPoissonArrivalProcess;
import umontreal.iro.lecuyer.contactcenters.dialer.Dialer;
import umontreal.iro.lecuyer.contactcenters.dialer.DialerPolicy;
import umontreal.iro.lecuyer.contactcenters.dialer.InfiniteDialerList;
import umontreal.iro.lecuyer.contactcenters.dialer.ThresholdDialerPolicy;
import umontreal.iro.lecuyer.contactcenters.queue.DequeueEvent;
import umontreal.iro.lecuyer.contactcenters.queue.QueueSizeStat;
```



```

import umontreal.iro.lecuyer.contactcenters.queue.StandardWaitingQueue;
import umontreal.iro.lecuyer.contactcenters.queue.WaitingQueue;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.contactcenters.router.QueuePriorityRouter;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.server.AgentGroup;
import umontreal.iro.lecuyer.contactcenters.server.AgentGroupSet;
import umontreal.iro.lecuyer.contactcenters.server.EndServiceEvent;
import umontreal.iro.lecuyer.contactcenters.server.GroupVolumeStat;
import umontreal.iro.lecuyer.probdist.GammaDist;
import umontreal.iro.lecuyer.randvar.ExponentialGen;
import umontreal.iro.lecuyer.randvar.GammaAcceptanceRejectionGen;
import umontreal.iro.lecuyer.randvar.GammaGen;
import umontreal.iro.lecuyer.randvar.RandomVariateGen;
import umontreal.iro.lecuyer.rng.MRG32k3a;
import umontreal.iro.lecuyer.rng.RandomStream;
import umontreal.iro.lecuyer.simevents.Sim;
import umontreal.iro.lecuyer.stat.FunctionOfMultipleMeansTally;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.stat.list.ListOfFunctionOfMultipleMeansTallies;
import umontreal.iro.lecuyer.stat.list.ListOfTallies;
import umontreal.iro.lecuyer.util.Chrono;
import umontreal.iro.lecuyer.util.RatioFunction;

public class Blend {
    // Contact type and agent group identifiers and names
    static final int INBOUND = 0;
    static final int OUTBOUND = 1;
    static final int BLEND = 1;
    // Input data (times are in minutes)
    static final String[] TYPENAMES = {
        "Inbound", "Outbound", "All types"
    };
    static final String[] GROUPNAMES = {
        "Inbound only", "Blend", "All groups"
    };
    // Input data (all times are in minutes)
    static final int K = 2;
    static final int I = 2;
    static final int P = 3;
    static final double PERIODDURATION = 120.0;
    static final double STARTDAY = 8*60;
    static final double STARTDIALER = 11*60;
    static final double[] ARRIVALRATES = {
        0, 68.45/30, 72.93/30, 71.92/30, 0
    };
    static final double ALPHA0 = 29.7;

```

```

static final double PROBBALK           = 0.005;
static final double INPATIENCETIME     = 500/60.0;
static final double ALPHA              = 0.755;
static final double BETA               = 753.8/60.0;
static final double AWT                = 20.0/60.0;
static final double OUTSERVICETIME     = 440.2/60.0;
static final double OUTPATIENCETIME    = 5.0/60.0;
static final double[] PROBBREACH = { 0.28, 0.28, 0.29, 0.29, 0.29 };
static final int MINFREEAGENTS = 4;
static final double KAPPA = 2.0;
static final int C = 0;
static final int[] INBOUNDAGENTS = { 0, 23, 23, 21, 21 };
static final int[] BLENDAGENTS = { 0, 16, 18, 16, 16 };
static final int QUEUECAPACITY = 80;
static final int[][] TYPETOGROUPMAP = {
    // Inbound calls
    { 0, 1 },
    // Outbound calls
    { 1 }
};
static final int[][] GROUPTOTYPEMAP = {
    // Inbound-only agents
    { 0 },
    // Blend agents
    { 1, 0 }
};
static final double LEVEL = 0.95; // Level of conf. int.
static final int NUMDAYS = 1000;

// Contact center components
PeriodChangeEvent pce; // Event marking beginning of periods
Dialer dialer;
Router router;
PiecewiseConstantPoissonArrivalProcess arrivProc;
AgentGroup inboundAgents;
AgentGroup blendAgents;
WaitingQueue inQueue;
WaitingQueue outQueue;

// Random number generators
// Busyness generator
RandomVariateGen bgen = new GammaGen
    (new MRG32k3a(), new GammaDist (ALPHA0, ALPHA0));
RandomStream streamBalk = new MRG32k3a();
// Service times for inbound calls
RandomVariateGen sigen = new GammaAcceptanceRejectionGen
    (new MRG32k3a(), new GammaDist (ALPHA, 1.0/BETA));

```

```

// Service times for outbound calls
RandomVariateGen sogen = new ExponentialGen (new MRG32k3a(), 1.0/OUTSERVICETIME);
// Patience times for inbound calls
RandomVariateGen pigen = new ExponentialGen (new MRG32k3a(), 1.0/INPATIENCETIME);

// Counters
int numTriedDialed;
double[] numArriv = new double[K + 1];
double[] numServed = new double[K + 1];
double[] numAbandoned = new double[K + 1];
double numBadSL;
double[] numPosWait = new double[K + 1];
double[] sumWaitingTimes = new double[K + 1];
QueueSizeStat inQueueSize;
QueueSizeStat outQueueSize;
GroupVolumeStat inboundVolume;
GroupVolumeStat blendVolume;
double[] Nb = new double[I + 1];
double[] N = new double[I + 1];
double[] qs = new double[K + 1];

// Statistical collectors
Tally triedDial = new Tally ("Number of tried outbound calls");
ListOfTallies<Tally> arrivals = create
    ("Number of arrived contacts", TYPENAMES);
ListOfTallies<Tally> served = create
    ("Number of served contacts", TYPENAMES);
ListOfTallies<Tally> abandoned = create
    ("Number of abandoned contacts", TYPENAMES);
Tally badSL = new Tally ("Number of contacts not in target");
FunctionOfMultipleMeansTally serviceLevel =
    new FunctionOfMultipleMeansTally (new RatioFunction(), "Service level", 2);
ListOfFunctionOfMultipleMeansTallies<FunctionOfMultipleMeansTally> occupancy =
    createRatio ("Agents' occupancy ratio", GROUPNAMES);
ListOfTallies<Tally> queueSize =
    create ("Time-average queue size", TYPENAMES);
ListOfFunctionOfMultipleMeansTallies<FunctionOfMultipleMeansTally> totalQueueWait =
    createRatio ("Waiting time", TYPENAMES);
ListOfTallies<Tally> posWait =
    create ("Number of contacts having to wait", TYPENAMES);

private ListOfTallies<Tally> create
(String name, String[] elementNames) {
    final ListOfTallies<Tally> lt =
        ListOfTallies.createWithTally (elementNames.length);
    lt.setName (name);
    for (int i = 0; i < elementNames.length; i++) {

```

```

        lt.get (i).setName (elementNames[i]);
        lt.get (i).setConfidenceIntervalStudent();
        lt.get (i).setConfidenceLevel (LEVEL);
    }
    return lt;
}

private ListOfFunctionOfMultipleMeansTallies<FunctionOfMultipleMeansTally>
createRatio (String name, String[] elementNames) {
    final RatioFunction ratio = new RatioFunction();
    final ListOfFunctionOfMultipleMeansTallies<FunctionOfMultipleMeansTally> mta =
        ListOfFunctionOfMultipleMeansTallies.create (ratio, 2, elementNames.length);
    mta.setName (name);
    for (int r = 0; r < elementNames.length; r++) {
        mta.get (r).setName (elementNames[r]);
        mta.get (r).setConfidenceIntervalDelta();
        mta.get (r).setConfidenceLevel (LEVEL);
    }
    return mta;
}

Blend() {
    pce = new PeriodChangeEvent (PERIODDURATION, P + 2, STARTDAY);
    pce.addPeriodChangeListener (new MyPeriodChangeListener());
    inboundAgents = new AgentGroup (pce, INBOUNDAGENTS);
    inboundVolume = new GroupVolumeStat (inboundAgents);
    blendAgents = new AgentGroup (pce, BLENDAGENTS);
    blendVolume = new GroupVolumeStat (blendAgents);

    inQueue = new StandardWaitingQueue();
    inQueueSize = new QueueSizeStat (inQueue);
    outQueue = new StandardWaitingQueue();
    outQueueSize = new QueueSizeStat (outQueue);

    final AgentGroupSet testSet = new AgentGroupSet();
    testSet.add (inboundAgents);
    testSet.add (blendAgents);
    final AgentGroupSet targetSet = new AgentGroupSet();
    targetSet.add (blendAgents);
    final DialerPolicy pol = new ThresholdDialerPolicy
        (new InfiniteDialerList (new MyContactFactory (OUTBOUND)),
         testSet, targetSet, MINFREEAGENTS, 1, KAPPA, C);

    final ConstantValueGenerator pReachGen = new ConstantValueGenerator
        (pce, K, PROBREACH);
    dialer = new Dialer (pol, new MRG32k3a(), pReachGen);
    final ContactCounter ntd = new ContactCounter();

```

```

    dialer.addReachListener (ntd);
    dialer.addFailListener (ntd);

    arrivProc = new PiecewiseConstantPoissonArrivalProcess
        (pce, new MyContactFactory (INBOUND), ARRIVALRATES,
         new MRG32k3a());

    router = new QueuePriorityRouter (TYPETOGROUPMAP, GROUPTOTYPEMAP);
    arrivProc.addNewContactListener (router);
    dialer.addReachListener (router);
    router.addExitedContactListener (new MyContactMeasures());
    router.setAgentGroup (INBOUND, inboundAgents);
    router.setAgentGroup (BLEND, blendAgents);
    router.getDialers (INBOUND).add (dialer);
    router.getDialers (BLEND).add (dialer);
    router.setWaitingQueue (INBOUND, inQueue);
    router.setWaitingQueue (OUTBOUND, outQueue);
    router.setTotalQueueCapacity (QUEUECAPACITY);
}

// Creates contacts
class MyContactFactory implements ContactFactory {
    int type;
    public MyContactFactory (int type) { this.type = type; }

    public Contact newInstance() {
        final Contact contact = new Contact (type);
        switch (type) {
            case INBOUND:
                // Inbound contact
                contact.setDefaultContactTime (sigen.nextDouble());
                final double u = streamBalk.nextDouble();
                if (u < PROBBALK) contact.setDefaultPatienceTime (0);
                else contact.setDefaultPatienceTime (pigen.nextDouble());
                break;
            case OUTBOUND:
                // Outbound contact
                contact.setDefaultContactTime (sogen.nextDouble());
                contact.setDefaultPatienceTime (OUTPATIENCE TIME);
                break;
        }
        return contact;
    }
}

// Updates counters
class MyContactMeasures implements ExitedContactListener {

```

```
public void blocked (Router router, Contact contact, int bType) {
    if (contact.getArrivalTime() < STARTDAY)
        return;
    final int type = contact.getTypeId();
    ++numArriv[type];
    ++numArriv[K];
    ++numAbandoned[type];
    ++numAbandoned[K];
    ++numPosWait[type];
    ++numPosWait[K];
}

public void dequeued (Router router, DequeueEvent ev) {
    final Contact contact = ev.getContact();
    if (contact.getArrivalTime() < STARTDAY)
        return;
    final int type = contact.getTypeId();
    ++numArriv[type];
    ++numArriv[K];
    ++numAbandoned[type];
    ++numAbandoned[K];
    final double qt = contact.getTotalQueueTime();
    sumWaitingTimes[type] += qt;
    sumWaitingTimes[K] += qt;
    ++numPosWait[type];
    ++numPosWait[K];
    if (qt >= AWT && type == INBOUND) ++numBadSL;
}

public void served (Router router, EndServiceEvent ev) {
    final Contact contact = ev.getContact();
    if (contact.getArrivalTime() < STARTDAY)
        return;
    final int type = contact.getTypeId();
    ++numArriv[type];
    ++numArriv[K];
    ++numServed[type];
    ++numServed[K];
    final double qt = contact.getTotalQueueTime();
    sumWaitingTimes[type] += qt;
    sumWaitingTimes[K] += qt;
    if (qt > 0) {
        ++numPosWait[type];
        ++numPosWait[K];
    }
    if (qt >= AWT && type == INBOUND) ++numBadSL;
}
```

```

}

// Counts tried outbound calls
class ContactCounter implements NewContactListener {
    public void newContact (Contact contact) {
        if (contact.getArrivalTime() < STARTDAY)
            return;
        ++numTriedDialed;
    }
}

// Ensures that integrals are computed over main periods only
class MyPeriodChangeListener implements PeriodChangeListener {
    public void changePeriod (PeriodChangeEvent pce) {
        if (pce.getCurrentPeriod() == 1) {
            // End of preliminary period
            inQueueSize.init();
            outQueueSize.init();
            inboundVolume.init();
            blendVolume.init();
        }
        else if (pce.getCurrentPeriod() == P + 1) {
            // Beginning of wrap-up period
            Nb[INBOUND] = inboundVolume.getStatNumBusyAgents().sum();
            N[INBOUND] = inboundVolume.getStatNumAgents().sum() +
                inboundVolume.getStatNumGhostAgents().sum();
            Nb[BLEND] = blendVolume.getStatNumBusyAgents().sum();
            N[BLEND] = blendVolume.getStatNumAgents().sum() +
                blendVolume.getStatNumGhostAgents().sum();
            Nb[2] = Nb[INBOUND] + Nb[BLEND];
            N[2] = N[INBOUND] + N[BLEND];

            qs[INBOUND] = inQueueSize.getStatQueueSize().sum()/(PERIODDURATION*P);
            qs[OUTBOUND] = outQueueSize.getStatQueueSize().sum()/(PERIODDURATION*P);
            qs[2] = qs[INBOUND] + qs[OUTBOUND];
        }
    }

    public void stop (PeriodChangeEvent pce) {}
}

void simulate (int n) {
    triedDial.init();
    arrivals.init();
    served.init();
    abandoned.init();
    badSL.init();
}

```

```

    serviceLevel.init();
    occupancy.init();
    queueSize.init();
    totalQueueWait.init();
    posWait.init();
    for (int r = 0; r < n; r++) simulateOneDay();
}

void simulateOneDay() {
    Sim.init();
    pce.init();
    arrivProc.init (bgen.nextDouble());
    dialer.init();
    inboundAgents.init();
    blendAgents.init();
    inQueue.init();
    outQueue.init();
    numTriedDialed = 0;
    numBadSL = 0;
    for (int k = 0; k <= K; k++) {
        numArriv[k] = 0;
        numServed[k] = 0;
        numAbandoned[k] = 0;
        numPosWait[k] = 0;
        sumWaitingTimes[k] = 0;
    }
    new ToggleEvent (arrivProc, true).schedule (STARTDAY - 5);
    new ToggleEvent (dialer, true).schedule (STARTDIALER);
    final double endDay = STARTDAY + P*PERIODDURATION;
    new ToggleEvent (dialer, false).schedule (endDay - 5);
    pce.start();
    Sim.start();
    pce.stop();
    addObs();
}

void addObs() {
    triedDial.add (numTriedDialed);
    arrivals.add (numArriv);
    served.add (numServed);
    abandoned.add (numAbandoned);
    badSL.add (numBadSL);
    serviceLevel.add (100*(numArriv[INBOUND] - numBadSL), numArriv[INBOUND]);
    for (int i = 0; i < Nb.length; i++)
        Nb[i] *= 100;
    occupancy.addSameDimension (Nb, N);
    queueSize.add (qs);
}

```



```

        totalQueueWait.addSameDimension (sumWaitingTimes, numArriv);
        posWait.add (numPosWait);
    }

    public void printStatistics() {
        System.out.println (triedDial.report ());
        System.out.println (arrivals.report ());
        System.out.println (served.report ());
        System.out.println (abandoned.report ());
        System.out.println (badSL.report ());
        System.out.println (serviceLevel.report ());
        System.out.println (occupancy.report ());
        System.out.println (queueSize.report ());
        System.out.println (totalQueueWait.report ());
        System.out.println (posWait.report ());
    }

    public static void main (String[] args) {
        final Blend blend = new Blend();
        final Chrono timer = new Chrono();
        blend.simulate (NUMDAYS);
        System.out.println ("CPU time: " + timer.format());
        blend.printStatistics();
    }
}

```

We took most input values from table 2 in [2], at periods 15, 16, and 17. From the input data, we get the expected number of arrivals per half hour. We then divide these rates by 30 to get the arrival rates per minute. The arrival process is constructed as in previous examples, using the processed arrival rates. The exponential patience time is set to 500 seconds, so we divide it by 60 to convert it in minutes. The probability of balking also comes from the input data and is set to 0.005 for the three observed periods. In the input data, service times follow a gamma distribution with shape parameter α and scale parameter $\beta = 1/\lambda$. Its mean, expressed in seconds, is given by $\alpha\beta$. To get the mean in minutes, we divide β by 60. When we construct the gamma generator, we will invert β for the parameter to be compatible with SSJ. The gamma service times are generated using acceptance-rejection, because inversion is too slow.

For the outbound type, we need the mean service time to be 440.2 seconds to follow the input data. In contrast with the input data in [2] setting the patience time for outbound calls to 0, we fixed this patience time to five seconds for some outbound calls to get queued. The probability of reaching a caller and the staffing vectors were all taken from the input data in [2].

To take balking into account, in the `newInstance` method of `MyContactFactory`, a uniform is drawn. If it is smaller than the probability of immediate abandonment, the patience

time is 0. Otherwise, it is exponential. For the outbound calls, the patience time is constant and set in the `newInstance` method too.

The router is a simple queue priority router with fixed tables. If an inbound call arrives, it is served by an inbound agent or, if no inbound agent is available, by a blend agent. If an outbound call arrives, it is served by a blend agent. If an inbound agent becomes free, it checks the inbound queue only. If a blend agent is free, it checks the outbound and the inbound queues, in that order. When agents arrive, they check for calls in the queues as when they become free.

The dialer is the newest and most complex part of this model. To be constructed, it requires at least a dialing policy, a random stream to determine if the right party is reached, and a value generator giving the probability of the right party to be reached. Reached calls are notified to the router while reached and failed calls are counted using a `ContactCounter` which is a custom new-contact listener. Failed calls include wrong party connects (answering machine or wrong person) as well as busy signals or no answer. The dialer is connected to the router in order to be automatically started (if it is enabled) each time any agent, whether inbound or blend, ends a service. It is also registered to be automatically initialized at the beginning of each replication.

When the dialer starts, if it is enabled, it uses its policy to determine how many calls to try. Dialed calls are extracted from a dialer list which is associated with the dialing policy. In most general settings, the list of outbound calls to try can depend on the state of the system. In this example, the dialer list is infinite, i.e., it uses a contact factory to produce a call each time it is asked to. Retrials of failed calls are included in the calls extracted from this infinite list. For each tried call, a random number is drawn to determine if it reaches the right party. If it is successful, it is sent to the router and the contact counter. Otherwise, it is sent to the contact counter only.

The dialer of this example uses a generic threshold-based policy which works as follows. It computes $N_F^T(t)$, the total number of free agents in a *test set* comprised in this example of both agent groups. It then checks that this number is greater than or equal to a certain threshold fixed to 4 in this example. If the condition holds, the dialer then computes $N_F^D(t)$, the number of free agents in a *target set*. In this example, this set contains blend agents only. If $N_F^D(t)$ is greater than or equal to a second threshold fixed to 1 in this example, the dialer tries to reach $2 * N_F^D(t)$ customers.

`inQueueSize` and `outQueueSize` are queue size statistical collector being used for computing the time-average queue size. Each queue size collector monitors a single waiting queue and computes the integrals for the queue size and optionally the number of contacts of each type in the queue. `inboundVolume` and `blendVolume` are agent group volume statistical collector used for the occupancy ratio.

Statistical collecting is performed the same way as in other examples. However, we want events to be counted for main periods only. As a result, events related to contacts arriving during the preliminary period are ignored. For agents' occupancy ratio and time-average queue size, we need to create a custom period-change listener. At the end of the preliminary, the agent group volume and queue size counters are reset to keep statistics about main

periods only. At the beginning of the wrap-up period, the necessary integrals are obtained before they are updated with observations from the wrap-up period. In addition, to get the time-average queue size over the opening hours, we divide the integral of queue size by the appropriate time duration. If we use the `average` method, the returned average is on all simulation time, including the complete preliminary period. Of course, our custom period-change listener needs to be registered with the period-change event to receive information; this operation is done in the constructor of `Blend`. Note that we could also solve this problem with an agent-group listener, as we did in section 4.3.

Listing 30 presents the statistical results of the call center.

Listing 30: Results of the program `Blend`

```

CPU time: 0:0:1.56
REPORT on Tally stat. collector ==> Number of tried outbound calls
  num. obs.      min      max      average      standard dev.
    1000      52.000    1670.000    1109.040      258.167

Report for Number of arrived contacts
  num obs.      min      max      average      std. dev.      conf. int.
Inbound    1000      425.000    1564.000     848.592     156.201    95.0% ( 838.899, 858.285)
Outbound   1000       21.000     475.000     321.896     74.396    95.0% ( 317.279, 326.513)
All types  1000      852.000    1586.000    1170.488     94.589    95.0% ( 1164.618, 1176.358)

Report for Number of served contacts
  num obs.      min      max      average      std. dev.      conf. int.
Inbound    1000      425.000    1434.000     843.477     149.608    95.0% ( 834.193, 852.761)
Outbound   1000       19.000     444.000     298.876     68.318    95.0% ( 294.637, 303.115)
All types  1000      825.000    1465.000    1142.353     93.386    95.0% ( 1136.558, 1148.148)

Report for Number of abandoned contacts
  num obs.      min      max      average      std. dev.      conf. int.
Inbound    1000       0.000     130.000       5.115      9.985    95.0% ( 4.495, 5.735)
Outbound   1000       0.000      49.000      23.020      7.609    95.0% ( 22.548, 23.492)
All types  1000       8.000     132.000     28.135      8.843    95.0% ( 27.586, 28.684)

REPORT on Tally stat. collector ==> Number of contacts not in target
  num. obs.      min      max      average      standard dev.
    1000       0.000     749.000     39.427      63.682

REPORT on Tally stat. collector ==> Service level
  func. of averages      standard dev.      num. obs.
    95.354           6.894           1000

Report for Agents' occupancy ratio
  func. of averages      std. dev.      nobs.      conf. int.
Inbound only           81.596       7.447      1000    95.0% ( 81.135, 82.058)
Blend                 55.665       6.900      1000    95.0% ( 55.237, 56.092)

```

All groups	70.514	6.818	1000	95.0% (70.091,	70.936)	
Report for Time-average queue size							
	num obs.	min	max	average	std. dev.	conf. int.	
Inbound	1000	0.000	2.967	0.115	0.234	95.0% (0.101, 0.130)
Outbound	1000	0.000	0.012	5.8E-3	1.9E-3	95.0% (5.7E-3, 5.9E-3)
All types	1000	4.8E-3	2.967	0.121	0.233	95.0% (0.107, 0.135)
Report for Waiting time							
	func. of averages	std. dev.	nobs.	conf. int.			
Inbound	0.049	0.094	1000	95.0% (0.043,	0.055)	
Outbound	6.5E-3	1.2E-3	1000	95.0% (6.4E-3,	6.6E-3)	
All types	0.037	0.070	1000	95.0% (0.033,	0.042)	
Report for Number of contacts having to wait							
	num obs.	min	max	average	std. dev.	conf. int.	
Inbound	1000	0.000	913.000	70.241	86.800	95.0% (64.855, 75.627)
Outbound	1000	0.000	57.000	27.445	9.034	95.0% (26.884, 28.006)
All types	1000	23.000	921.000	97.686	81.085	95.0% (92.654, 102.718)

References

- [1] A. Deslauriers. Modélisation et simulation d'un centre d'appels téléphoniques dans un environnement mixte. Master's thesis, Department of Computer Science and Operations Research, University of Montreal, Montreal, Canada, 2003.
- [2] A. Deslauriers, J. Pichitlamken, P. L'Ecuyer, A. Ingolfsson, and A. N. Avramidis. Markov chain models of a telephone call center with call blending. *Computers and Operations Research*, 34(6):1616–1645, 2007.
- [3] D. Flanagan. *Java in a Nutshell*. O'Reilly, Sebastopol, CA, third edition, 1999.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, second edition, 1998.
- [5] N. Gans, G. Koole, and A. Mandelbaum. Telephone call centers: Tutorial, review, and research prospects. *Manufacturing and Service Operations Management*, 5:79–141, 2003.
- [6] Wolfgang Hoschek. *The Colt Distribution: Open Source Libraries for High Performance Scientific and Technical Computing in Java*. CERN, Geneva, 2004. Available at <http://dsd.lbl.gov/~hoschek/colt/>.
- [7] L. Kleinrock. *Queueing Systems, Vol. 1*. Wiley, New York, NY, 1975.
- [8] P. L'Ecuyer. *SSJ: A Java Library for Stochastic Simulation*, 2004. Software user's guide, available at <http://www.iro.umontreal.ca/~lecuyer>.
- [9] V. Mehrotra and J. Fama. Call center simulation modeling: Methods, challenges, and opportunities. In *Proceedings of the 2003 Winter Simulation Conference*, pages 135–143. IEEE Press, 2003.
- [10] R. J. Serfling. *Approximation Theorems for Mathematical Statistics*. Wiley, New York, NY, 1980.