

User's Guide for ContactCenters Simulation Library

Core API Specification

Version: March 4, 2014

ERIC BUIST

This is the API specification of the ContactCenters library, a software library providing a toolset to build contact center simulators. The document provides detailed documentation about each interface, class and method for one to extend existing contact center simulators or create new ones.

Contents

Overview	2
Package umontreal.iro.lecuyer.contactcenters	4
Initializable	5
Named	6
ToggleElement	7
ToggleEvent	8
SwitchEvent	10
PeriodChangeListener	12
PeriodChangeEvent	13
NonStationaryMeasureMatrix	21
MultiPeriodGen	22
ValueGenerator	29
ConstantValueGenerator	30
RandomValueGenerator	32
MinValueGenerator	34
ContactCenter	37
MatrixUtil	40
RandomStreamUtil	42
StatUtil	44
RepSimCC	46
BatchMeansSimCC	51

Package umontreal.iro.lecuyer.contactcenters.contact	54
Contact	55
ServiceTimes	67
ContactFactory	70
SimpleContactFactory	71
SingleTypeContactFactory	73
RandomTypeContactFactory	78
ContactInstantiationException	80
NewContactListener	82
ContactSumMatrix	83
ContactStepInfo	85
TrunkGroup	86
ContactSource	89
ContactArrivalProcess	91
StationaryContactArrivalProcess	98
PoissonArrivalProcess	99
PiecewiseConstantPoissonArrivalProcess	103
PoissonArrivalProcessWithTimeIntervals	108
PoissonGammaArrivalProcess	110
PoissonGammaNortaRatesArrivalProcess	114
GammaParameterEstimator	118
DirichletCompoundArrivalProcess	128
PoissonUniformArrivalProcess	131
FixedCountsArrivalProcess	134
DirichletArrivalProcess	135
NORTADrivenArrivalProcess	139
CorrelationMatrixCorrector	142
CorrelationMtxFitting	144
PoissonArrivalProcessWithInversion	145
PoissonArrivalProcessWithThinning	147

Package umontreal.iro.lecuyer.contactcenters.queue	149
WaitingQueue	150
DequeueEvent	159
DequeueEventComparator	162
WaitingQueueSet	163
WaitingQueueListener	165
StandardWaitingQueue	166
PriorityWaitingQueue	167
QueueWaitingQueue	168
QueueSizeStat	169
QueueSizeStatMeasureMatrix	171
ContactPatienceTimeGenerator	173
WaitingQueueState	174
EnqueueEvent	175
 Package umontreal.iro.lecuyer.contactcenters.queuemodel	 177
ErlangC	178
 Package umontreal.iro.lecuyer.contactcenters.server	 184
AgentGroup	185
EndServiceEvent	196
DetailedAgentGroup	200
EndServiceEventDetailed	206
Agent	207
AgentGroupSet	212
AgentGroupListener	215
AgentListener	216
GroupVolumeStat	218
GroupVolumeStatMeasureMatrix	221
ContactTimeGenerator	225
AfterContactTimeGenerator	227
AgentGroupState	229
DetailedAgentGroupState	230
AgentState	231
StartServiceEvent	232
SetNumAgentsEvent	236
RestoreAgentsEvent	237

Package umontreal.iro.lecuyer.contactcenters.dialer	238
Dialer	239
DialerActionEvent	247
DialerPolicy	248
ConstantDialerPolicy	250
ThresholdDialerPolicy	251
BadContactMismatchRatesDialerPolicy	254
AgentsMoveDialerPolicy	259
DialerList	267
InfiniteDialerList	269
ContactListenerDialerList	270
DialerListNoQueueing	272
DialerState	273
DialerActionState	274
MismatchChecker	275
 Package umontreal.iro.lecuyer.contactcenters.router	 276
Router	278
WaitingQueueType	295
WaitingQueueStructure	296
ContactReroutingEvent	297
AgentReroutingEvent	298
QueuePriorityRouter	299
QueueAtLastGroupRouter	302
LongestQueueFirstRouter	306
SingleFIFOQueueRouter	307
LongestWeightedWaitingTimeRouter	308
AgentsPrefRouter	309
AgentsPrefRouterWithDelays	319
AgentSelectionScore	324
ContactSelectionScore	325
LocalSpecRouter	326
QueueRatioOverflowRouter	331

ExpDelayRouter	335
OverflowAndPriorityRouter	337
RankFunction	343
RoutingStageInfo	344
ExitedContactListener	345
RoutingTableUtils	346
AgentGroupSelectors	355
RouterState	358
ReroutingState	359
EnqueueEventWithRerouting	360
Package umontreal.iro.lecuyer.contactcenters.expdelay	362
WaitingTimePredictor	363
ExpectedDelayPredictor	365
LastWaitingTimePredictor	366
LastWaitingTimePerQueuePredictor	367
HeadOfQueuePredictor	368
Package umontreal.iro.lecuyer.stat.mperiods	369
MeasureMatrix	371
StatProbeMeasureMatrix	373
ListOfStatProbesMeasureMatrix	375
MatrixOfStatProbesMeasureMatrix	377
MeasureSet	378
SumMatrix	382
SumMatrixSW	385
IntegralMeasureMatrix	388
IntegralMeasureMatrixSW	391
Package umontreal.iro.lecuyer.simevents	393
SimTimeMeasureMatrix	394
UnusableSimulator	395

Overview

A *contact center* is a set of resources (communication equipment, employees, computers, etc.) providing an interface between customers and a business [1, 8, 15, 4]. A *contact* represents a customer's request for some service such as information, subscription, order, etc. Customers may use various media for contacting a business: telephone, fax, mail, or Internet. A contact center processing phone calls only is named a *call center*.

Inbound contacts are initiated by customers trying to communicate with the business. A customer can be *blocked*, i.e., receive a busy signal, if all phone lines are used at the time he calls. He can also be queued if service cannot be started immediately. A queued customer may become impatient and abandon without receiving service. A *retrial* occurs if the customer having abandoned tries to contact the business again. A served customer may also *return* to get new service, or to satisfy its initial request.

Outbound contacts are initiated by agents contacting customers, or by a *predictive dialer* making phone calls by trying to anticipate the number of free agents at the time contacted customers are reached. A *right party connect* occurs when an outbound contact is successful, i.e., the right person has been reached. A *mismatch* represents a successful contact that cannot be served immediately. Often, these mismatches are considered as lost calls, because most customers will not wait after they answer.

Modern contact centers use *skill-based* routing for processing different types of requests when each agent is trained for handling only a subset of these types. Each contact is assigned a type (or skill) k in $0, \dots, K - 1$. To determine this type, before reaching an agent, a customer must indicate his needs: callers interact with an *interactive voice response* (IVR) unit while Internet users enter data in a Web form. Outbound contacts can also have a type, since all customers are not contacted for the same reason.

The agents are partitioned in I agent groups or skill sets. All agents in a group i share the same skills, i.e., they can serve the same types of contacts (although some members may be more efficient than others).

Queueing theory can be used to derive approximations for estimating the performance measures of contact centers, but only for models that oversimplify the complexities of real-life systems for which only simulation can provide accurate results. Simulation permits the analysis of the impact of parameter changes on contact center's performance. For example, it can evaluate service level of contacts, occupancy ratio of agents, waiting times, etc. for (almost) arbitrary contact centers.

The *ContactCenters* library provides a set of building blocks to help programmers in the development of contact center simulators. The library uses Stochastic Simulation in Java (SSJ) [14] to perform discrete-event simulation and to generate random variates. It also relies on Collections Tuned (Colt) [9] for matrix manipulation.

A precompiled generic contact center simulator, adapted for blend and multi-skill models, is provided and can use XML files for parameters. See `guidemsk.pdf` for more information about this simulator, how to configure it, and how to use it. The document `guideapp.pdf`

describes in more details the various interfaces, classes, and methods permitting the user to access precompiled simulators from other Java programs.

For existing simulators to be extended or new ones to be created, the simulation toolset provided by this library must be used directly. This toolset is comprised of various components grouped in different packages. Each component corresponds to a specific contact center element and can easily be extended or replaced by the user. The `umontreal.iro.lecuyer.contactcenters` package provides some facilities to manage contact centers in general. It defines base classes for contact center simulation applications as well as a framework to generate contact-specific values during the simulation. The package `umontreal.iro.lecuyer.contactcenters.contact` defines the `Contact` class whose instances represent the contacts traveling into the system. It also defines several arrival processes for inbound contacts.

The package `umontreal.iro.lecuyer.contactcenters.server` defines the facilities for serving contacts by agents. It defines agent groups as well as a data structure to store information about served contacts.

The package `umontreal.iro.lecuyer.contactcenters.queue` defines the waiting queue contacts can have to wait in if they cannot be served at the time they enter the center. It defines a First-In-First-Out waiting queue as well as a generic priority queue, with a data structure to store information about queued contacts.

The package `umontreal.iro.lecuyer.contactcenters.dialer` defines the dialer capable of performing outbound calls. The defined dialer can implement a complex dialing policy obtaining contacts from various sources.

The package `umontreal.iro.lecuyer.contactcenters.router` implements the routing facilities, linking all the contact center objects together. It defines a base class representing a router as well as several subclasses for various routing policies.

Package `umontreal.iro.lecuyer.contactcenters`

Contains general interfaces and classes for simulating contact centers. The interfaces `Named` and `Initializable` are defined to represent objects having a name and which can be initialized, respectively. The interface `ToggleElement` is defined for contact center objects that can be enabled or disabled. The simulation event `ToggleEvent` can be used to toggle objects implementing `ToggleElement`.

The class `PeriodChangeEvent` can be use to divide the simulation time into periods to simulate non-stationary contact centers. The interface `PeriodChangeListener` can be implemented by simulation objects to be notified when period changes occur.

The `ValueGenerator` interface is defined to generate state-dependent random values during the contact center simulation. Some general-purpose implementations of this interface are provided.

The package also contains a base `ContactCenter` class providing convenience methods used to implement a simulator.

Initializable

Defines an object that can be initialized by the use of an initialization method.

```
package umontreal.iro.lecuyer.contactcenters;  
  
public interface Initializable
```

Method

```
    public void init()  
        Initializes this object.
```

Named

Represents an object having a name.

```
package umontreal.iro.lecuyer.contactcenters;  
  
public interface Named
```

Methods

```
public String getName()
```

Returns the name associated with this object. If no name was set, this must return an empty string, not `null`.

Returns the name of this object.

```
public void setName (String name)
```

Sets the name of this object to `name`. The given name cannot be `null` and the implementation can throw an `UnsupportedOperationException` if the name is read-only.

Parameter

`name` the new name of the object.

Throws

`UnsupportedOperationException` if the name cannot be changed.

`NullPointerException` if `name` is `null`.

ToggleElement

Specifies an element that can be enabled or disabled at any time during the simulation. The meaning of the “enabled” and “disabled” states depends on the particular toggle element. For example, an enabled contact arrival process provides contacts to the system whereas a disabled arrival process does not.

```
package umontreal.iro.lecuyer.contactcenters;
```

```
public interface ToggleElement
```

Methods

```
public void start()
```

Enables the element represented by this object. This method throws an `IllegalStateException` if the element is already enabled.

Throws

`IllegalStateException` if the element is already enabled.

```
public void stop()
```

Disables the element represented by this object. This method throws an `IllegalStateException` if the element is already disabled.

Throws

`IllegalStateException` if the element is already disabled.

```
public boolean isStarted()
```

Determines if the element is enabled or disabled. Returns `true` if the element is enabled, `false` otherwise.

Returns the current state of the element.

ToggleEvent

This event instructs a toggle element, i.e., any object implementing `ToggleElement`, to be started or stopped during the simulation. It can be useful to toggle some elements of a contact center, e.g., arrival processes, at determined moments during simulation. After the event is constructed and scheduled, when the simulation clock reaches the scheduled time, the event starts or stops the associated toggle element. Note that for the event to have a meaningful name when printing the event list, the target toggle element should override its `toString` method.

```
package umontreal.iro.lecuyer.contactcenters;
```

```
public class ToggleEvent extends Event
```

Constructors

```
public ToggleEvent (ToggleElement element, boolean start)
```

Constructs a new toggle event that will, at the time of its execution, start the toggle element `element` if `start` is `true`, or stop it if `enabled` is `false`.

Parameters

`element` the toggle element affected by the event.

`start` the status of the toggle element after the event occurs.

Throws

`NullPointerException` if `element` is `null`.

```
public ToggleEvent (Simulator sim, ToggleElement element, boolean start)
```

Equivalent to `ToggleEvent (ToggleElement, boolean)`, with a user-defined simulator `sim`.

Parameters

`sim` the simulator associated with the toggle event.

`element` the toggle element affected by the event.

`start` the status of the toggle element after the event occurs.

Throws

`NullPointerException` if `sim` or `element` are `null`.

Methods

```
public ToggleElement getToggleElement()
```

Returns the toggle element affected by this event.

Returns the toggle element affected by this event.

```
public void setToggleElement (ToggleElement element)
```

Changes the associated toggle element to **element**.

Parameter

element the new toggle element.

Throws

`NullPointerException` if **element** is `null`.

```
public boolean getStart()
```

Returns the status of the toggle element associated with this object after this event has occurred.

Returns the status of the associated toggle element after this event has occurred.

```
public void setStart (boolean start)
```

When the event occurs, the activity status of the toggle element will be set to **start**.

Parameter

start `true` if the toggle element will be started, `false` if it will be stopped.

SwitchEvent

Represents an event that toggles an element on predefined simulation times. This differs from **ToggleEvent** that occurs only at a specific simulation time, and enables or disables the toggle element once. This event is constructed with a toggle element, and an array of simulation times. The constructor determines the first time, in the array, that is greater than the current simulation time, and schedules the event at that time. When the event happens, the element is toggled, and the event is scheduled again until all the times in the array have been used.

```
package umontreal.iro.lecuyer.contactcenters;
```

```
public class SwitchEvent extends Event
```

Constructors

```
public SwitchEvent (ToggleElement el, double[] times)
```

Constructs a new switch event from the toggle element **el**, and the simulation times **times**. The times in **times** must be sorted in ascending order.

Parameters

el the toggle element.

times the simulation times the event will occur.

Throws

`NullPointerException` if **el** or **times** are null.

```
public SwitchEvent (Simulator sim, ToggleElement el, double[] times)
```

Equivalent to `SwitchEvent (ToggleElement, double[])`, with a user-defined simulator **sim**.

Parameters

sim the simulator attached to the new event.

el the toggle element.

times the simulation times the event will occur.

Throws

`NullPointerException` if **sim**, **el**, or **times** are null.

Methods

```
public ToggleElement getToggleElement()
```

Returns the toggle element affected with this event.

Returns the affected toggle element.

```
public double[] getToggleTimes()
```

Returns an array containing the toggle times used by this event.

Returns the array of toggle times.

```
public void skipTime()
```

Cancels this event if it is scheduled, and skips to the next toggle time.

PeriodChangeListener

Represents a period-change listener being notified when period-change events occur.

```
package umontreal.iro.lecuyer.contactcenters;  
  
public interface PeriodChangeListener
```

Methods

```
public void changePeriod (PeriodChangeEvent pce)
```

Switches to the next period defined by `pce`. This can change the parameters and correct scheduled events accordingly. If no parameters are available for the new period, the method should try to use those of the last available period. The listener is called after the period change has occurred, so `PeriodChangeEvent.getCurrentPeriod()` returns the index of the new period.

Parameter

`pce` the source period-change event.

```
public void stop (PeriodChangeEvent pce)
```

This method is called after the period-change event is stopped by `PeriodChangeEvent.stop()`.

Parameter

`pce` the period-change event being stopped.

PeriodChangeEvent

Defines a simulation event that occurs upon period changes and supporting fixed-sized or variable-sized periods. Because this event must cover the complete simulation horizon (day, week, etc.), not only the times at which the contact center is opened, three types of periods need to be defined.

The P periods during which the contact center is opened are denoted *main periods*, or simply periods. Main period p , where $p = 1, \dots, P$, corresponds to simulation time interval $[t_{p-1}, t_p[$, where $t_0 < \dots < t_P$. During the *preliminary period* $[0, t_0)$, the contact center is closed. Sometimes, arrivals start at time $t_0 - \tau$ for a queue to build up before agents enter into service. During the *wrap-up period* $[t_P, T]$, no more arrival occurs, but ongoing services are terminated. Note that preliminary and wrap-up periods are more useful when a simulation replication corresponds to a day.

Before starting the simulation, the period-change event should be initialized by calling `init()`, which resets the current period index. The event needs to be started by using `start()`; this schedules it at the beginning of the first main period. This also schedules auxiliary event managing period changes at the other periods. It is recommended to start the period-change event before scheduling any other event to ensure that period changes have priority over other events. When the period change occurs, it is notified to any registered `PeriodChangeListener` implementation. When returning from `Sim.start` or just before calling `Sim.stop`, it is recommended to call the `stop()` method of this object, since the end of the wrap-up period is not scheduled as an event. This notifies all registered listeners about the end of the simulation, and disables any remaining auxiliary event. This can be useful for some statistical collectors.

Note: the `PeriodChangeListener` implementations are notified in the order of the list returned by `getPeriodChangeListeners()`, and a period-change listener modifying the list of listeners by using `addPeriodChangeListener (PeriodChangeListener)` or `removePeriodChangeListener (PeriodChangeListener)` could result in unpredictable behavior.

```
package umontreal.iro.lecuyer.contactcenters;

public class PeriodChangeEvent extends Event
    implements Initializable, Named, ToggleElement
```

Field

```
public static final double PRIORITY
```

Default priority of period-change events. This priority index is set to 0.1, because the period-change event should execute before any other event at the same time.

Constructors

```
public PeriodChangeEvent (double periodDuration, int numPeriods, double
                        startingTime)
```

Constructs a new period-change event with fixed-sized main periods of duration `periodDuration`, a total of $P + 2 = \text{numPeriods}$ periods, with the first main period beginning at time $t_0 = \text{startingTime}$, and using the default simulator. For the event to be used, there must be at least two periods (preliminary and wrap-up). With a total of `numPeriods` periods, the event defines `numPeriods - 2` main periods, a preliminary, and a wrap-up periods, even if the given starting time is 0.

Note that this constructor calls the `setFixedPeriods (double, double)` method.

Parameters

`periodDuration` the length of each period, in simulation time units.

`numPeriods` the total number of periods $P + 2$.

`startingTime` the beginning of the first period.

Throws

`IllegalArgumentException` if the number of periods is smaller than 2 or the period duration or starting time is smaller than 0.

```
public PeriodChangeEvent (Simulator sim, double periodDuration, int
                        numPeriods, double startingTime)
```

Equivalent to `PeriodChangeEvent (double, int, double)`, with a user-defined simulator `sim`.

Parameters

`sim` the simulator attached to the new event.

`periodDuration` the length of each period, in simulation time units.

`numPeriods` the total number of periods $P + 2$.

`startingTime` the beginning of the first period.

Throws

`IllegalArgumentException` if `sim` is null, if the number of periods is smaller than 2, or the period duration or starting time is smaller than 0.

```
public PeriodChangeEvent (double... endingTimes)
```

Constructs a new period-change event with variable-sized periods, using the default simulator. The object will support `endingTimes.length + 1` periods where $t_p = \text{endTimes}[p]$. The ending times in the array must be non-decreasing, otherwise an `IllegalArgumentException` is thrown.

This constructor accepts a variable number of arguments, i.e., one can use `new PeriodChangeEvent (t1, t2, t3, t4, t5)`, where `tN` are ending times. One can also pass a regular array.

Note that this constructor calls the `setEndingTimes (double...)` method.

Parameter

`endingTimes` the ending times of periods.

Throws

`IllegalArgumentException` if one ending time is negative or the ending times are not non-decreasing.

```
public PeriodChangeEvent (Simulator sim, double... endingTimes)
    Equivalent to PeriodChangeEvent (double...), with a user-defined simulator sim.
```

Parameters

`sim` the simulator attached to this event.

`endingTimes` the ending times of periods.

Throws

`IllegalArgumentException` if `sim` is null, or if one ending time is negative or the ending times are not non-decreasing.

Methods

```
public final void setFixedPeriods (double periodDuration, double
                                   startingTime)
```

Sets this period-change event to fixed-length periods of duration `periodDuration`, and with main period starting at $t_0 = \text{startingTime}$. It should not be used during a simulation replication. It is not allowed to modify the number of periods as many other objects depend on this parameter.

Parameters

`periodDuration` the length of each main period.

`startingTime` the starting time of the first main period.

Throws

`IllegalArgumentException` if one argument is negative.

```
public final void setEndingTimes (double... endingTimes)
```

Changes the ending times of periods to `endingTimes`. This should not be used during a simulation replication, otherwise the period-change event will be cancelled and no more period change will be notified to listeners. It is also not allowed to change the number of periods, because many objects can depend on this number.

Parameter

`endingTimes` the new period ending times.

Throws

`IllegalArgumentException` if one ending time is negative, the ending times are not non-decreasing, or one tries to change the number of periods.

```
public void addPeriodChangeListener (PeriodChangeListener l)
```

Registers the period-change listener `l` to be notified when a period change occurs.

Parameter

`l` the listener to be registered.

Throws

`NullPointerException` if `l` is null.

```
public void removePeriodChangeListener (PeriodChangeListener l)
```

Removes the period-change listener `l` from this period-change event.

Parameter

`l` the period change listener being removed.

```
public void clearPeriodChangeListeners()
```

Removes all the period change listeners from this event.

```
public List<PeriodChangeListener> getPeriodChangeListeners  
( )
```

Returns an unmodifiable list containing the period-change listeners currently registered with this event.

Returns the list of period-change listeners.

```
public void init()
```

Equivalent to `init (simulator().time())`.

```
public void init (double initTime)
```

Initiates this period-change event at initial time `initTime`. This initializes the event for a new simulation replication, which resets the current period index. When `start()` is called, no period changes will be scheduled before `initTime`.

Parameter

`initTime` the initialization time.

```
public void initAndNotify()
```

Equivalent to `initAndNotify (simulator().time())`.

```
public void initAndNotify (double initTime)
```

Calls `init()` and notifies the period-change listeners if the period changed due to the initialization. This can be useful to force period-change listeners to restore parameters.

```
public void start()
```

Starts the period-change event by scheduling it.

```
public void stop()
```

This method should be called when the simulation ends. It calls the `PeriodChangeListener.stop (PeriodChangeEvent)` method of all registered `PeriodChangeListener` implementations.

```
public boolean wasStopped()
```

Determines if this period-change event was stopped since the last call to `init()`.

Returns `true` if and only if the period-change event was stopped.

```
public int getCurrentPeriod()
```

Returns the index of the current simulation period.

Returns the index of the current period.

```
public void setCurrentPeriod (int p)
```

Sets the current period to `p` and disables all period changes initiated by this event. When the period is arbitrarily set by this method, the period-change event is cancelled and cannot be used to change period until the next call to `init()`, or `initAndNotify()`. However, this method can be used multiple times without calling `init()`. Each time this method changes the current period, registered period-change listeners are notified.

Parameter

`p` the new period index.

Throws

`IllegalArgumentException` if the period index is negative or greater than or equal to `getNumPeriods()`.

```
public boolean isLockedPeriod()
```

Returns `true` if the current period was changed using `setCurrentPeriod (int)` from the last call to `init()`. When the period is locked, only calls to `setCurrentPeriod (int)` can change the period index. If the period is not locked, this returns `false`.

Returns the period locking indicator.

```
public int getCurrentMainPeriod()
```

Returns the current main period for this period-change event. This is equivalent to `getMainPeriod (getCurrentPeriod())`.

Returns the index of the current main period.

```
public boolean isPreliminaryPeriod (int period)
```

Determines if the period index `period` corresponds to the preliminary period. This method returns `true` if and only if `period` is equal to 0.

Parameter

`period` the tested period index.

Returns `true` if and only if the period index corresponds to the preliminary period.

```
public boolean isMainPeriod (int period)
```

Determines if the period index `period` corresponds to a main period. The method returns `true` if and only if `period` is greater than 0 and smaller than or equal to `getNumPeriods() - 2`.

Parameter

`period` the tested period index.

Returns `true` if `period` corresponds to a main period.

```
public boolean isWrapupPeriod (int period)
```

Determines if the period index `period` corresponds to the wrap-up period. This method returns `true` if and only if `period` is equal to `getNumPeriods() - 1`.

Parameter

`period` the tested period index.

Returns `true` if and only if the period index corresponds to the wrap-up period.

```
public int getMainPeriod (int period)
```

Returns the main period index corresponding to period `period`. This returns the result of `period - 1` for main periods. If the period is the preliminary period, this returns 0, the index of the first main period. If the period is the wrap-up period, this returns the index of the last main period.

Parameter

`period` the period index to be processed.

Returns the main period index.

```
public int getPeriod (double simTime)
```

Computes the period index corresponding to the simulation time `simTime`.

Parameter

`simTime` the simulation time.

Returns the corresponding period index.

`public boolean isPeriodStartingTime (double time)`

Determines if the time `time` corresponds to the beginning of a period. This class cannot force period-change events to have priority over simulation events happening at the time of a period change, but the period change should usually be processed before any other event happening at the same time. Otherwise, parameters may have inconsistent values. This method can be used to help reschedule offending events manually if they cannot be scheduled after `start()` is called. One can use `getPeriod (double)` to obtain the period corresponding to the given simulation time if needed.

Parameter

`time` the simulation time to test.

Returns `true` if the given time corresponds to the time of a (future) period change, `false` otherwise.

`public double getPeriodStartingTime (int period)`

Returns the simulation time at which the period `period` starts.

Parameter

`period` the index of the queried period.

Returns the simulation time at which the period begins.

Throws

`IllegalArgumentException` if the period index is invalid.

`public double getPeriodEndingTime (int period)`

Returns the simulation time at which the period `period` ends. If the index of the last period is given, this returns the time at which `stop()` was called. If it was not called yet, this returns the current simulation time if the current period is the last one or `Double.NaN` otherwise.

Parameter

`period` the queried period.

Returns the simulation time of the beginning of the period.

Throws

`IllegalArgumentException` if the period index is invalid.

`public double getPeriodDuration (int period)`

Returns the duration of the period `period`. This corresponds to `getPeriodEndingTime (int)` minus `getPeriodStartingTime (int)`.

Parameter

`period` the period of interest.

Returns the duration of the period.

Throws

`IllegalArgumentException` if the period index is invalid.

```
public int getNumPeriods()
```

Returns $P + 2$, the number of periods supported by this period change event.

Returns the number of periods.

```
public int getNumMainPeriods()
```

Returns P , the number of main periods used by this period change event, i.e., `getNumPeriods() - 2`.

Returns the number of main periods.

NonStationaryMeasureMatrix

Computes per-period values for a one-period measure matrix. This class extends the `IntegralMeasureMatrix` and maps a period with a contact center period. It automatically calls `IntegralMeasureMatrix.newRecord()` upon period changes.

```
package umontreal.iro.lecuyer.contactcenters;

public class NonStationaryMeasureMatrix<M extends MeasureMatrix> extends
    IntegralMeasureMatrix<M>
    implements PeriodChangeListener
```

Constructor

```
public NonStationaryMeasureMatrix (PeriodChangeEvent pce, M mat)
```

Constructs a new non-stationary measure matrix from the one-period measure matrix `mat` and using the period change event `pce` to define the periods.

Parameters

`pce` the period change event.

`mat` the one-period only measure matrix.

Throws

`IllegalArgumentException` if a multiple-periods measure matrix is given.

`NullPointerException` if any argument is null.

MultiPeriodGen

Represents a random variate generator for non-stationary distributions with constant parameters during each period. When a new random variate is required, a random variate generator corresponding to the appropriate period is selected and a value is drawn from this generator.

This generator supports caching by using internal `RandomVariateGenWithCache` instances for each period. If a single cache was used, the generator could recover a value whose distribution does not correspond with the current period. Caching is disabled by default, and can be enabled by using the `setCaching (boolean)` method.

```
package umontreal.iro.lecuyer.contactcenters;

public class MultiPeriodGen extends RandomVariateGen
    implements ValueGenerator
```

Constructors

```
public MultiPeriodGen (PeriodChangeEvent pce, RandomVariateGen gen)
```

Constructs a new multi-period random variate generator with period-change event `pce`, and random variate generator `gen` for every period.

Parameters

`pce` the period-change event.

`gen` one random variate generator for every period.

Throws

`NullPointerException` if any argument is `null`.

```
public MultiPeriodGen (PeriodChangeEvent pce, RandomVariateGen[] gens)
```

Constructs a new multi-period random variate generator with period-change event `pce`, and the per-period random variate generators `gens`.

Parameters

`pce` the period change event.

`gens` one random variate generator for each period.

Throws

`NullPointerException` if any argument is `null`.

`IllegalArgumentException` if the length of `gens` does not correspond to the number of periods.

Methods

`public boolean isCaching()`

Determines if this multiple-periods generator is caching the generated values, using internal `RandomVariateGenWithCache` objects. By default, caching is disabled for better memory utilization.

Returns the status of the caching.

`public void setCaching (boolean caching)`

Sets the status of the caching for this generator.

Parameter

`caching` the new status of the caching.

`public PeriodChangeEvent getPeriodChangeEvent()`

Returns the period-change event associated with this object.

Returns the associated period-change event.

`public RandomVariateGen[] getGenerators()`

Returns the random variate generators associated with this object.

Returns the associated random variate generators.

`public RandomVariateGenWithCache[] getGeneratorsWithCache
()`

Returns the random variate generators with cache used by this object. If caching is disabled (the default), this method throws an `IllegalStateException`.

Returns the random variate generators with cache.

Throws

`IllegalStateException` if caching is disabled.

`public void setGenerators (RandomVariateGen[] gens)`

Sets the per-period random variate generators to `gens`. Note that if caching is enabled, the cache is reset when using this method.

Parameter

`gens` the array containing the new random variate generators.

Throws

`IllegalArgumentException` if the length of `gens` is invalid.

`NullPointerException` if any argument is `null`.

`public RandomVariateGen getGenerator (int p)`

Returns the random variate generator corresponding to the period `p`.

Parameter

`p` index of the period.

Returns the corresponding random variate generator.

Throws

`ArrayIndexOutOfBoundsException` if `p` is out of bounds.

```
public RandomVariateGenWithCache getGeneratorWithCache (int p)
```

Returns the random variate generator with cache corresponding to the period `p`. If caching is disabled (the default), this method throws an `IllegalStateException`.

Parameter

`p` index of the period.

Returns the corresponding random variate generator with cache.

Throws

`ArrayIndexOutOfBoundsException` if `p` is out of bounds.

`IllegalStateException` if caching is disabled.

```
public void setGenerator (int p, RandomVariateGen gen)
```

Sets the random variate generator for period `p` to `gen`.

Parameters

`p` the period index.

`gen` the new random variate generator.

Throws

`ArrayIndexOutOfBoundsException` if `p` is out of bounds.

```
public void initCache()
```

Resets the cache of this generator, if caching is enabled. If caching is disabled, this method does nothing. When the cache is reset, cached values are returned upon calls to `nextDouble()`, until the cache is exhausted. When there is no more cached value, random variates are computed as usual.

```
public void clearCache()
```

Clears the values cached by this generator. If caching is disabled, this method does nothing.

```
public int[] getCacheIndices()
```

Returns an array containing the cache indices of each per-period generator.

Returns the array of cache indices.

Throws

`IllegalStateException` if caching is disabled.

```
public void setCacheIndices (int[] ind)
```

Sets the array of cache indices to `ind`.

Parameter

`ind` the new array of cache indices.

Throws

`NullPointerException` if `ind` is `null`.

`IllegalArgumentException` if `ind` has incorrect size.

`IllegalStateException` if caching is disabled.

```
public DoubleArrayList[] getCachedValues()
```

Returns an array of array lists containing the values cached by each period-specific generator.

Returns the array of cached values.

```
public void setCachedValues (DoubleArrayList[] values)
```

Sets the array list containing the cached values to `values[g]` for each period-specific generator `g`. This resets the cache index to the size of the given array for each generator.

Parameter

`values` the array list of cached values.

Throws

`NullPointerException` if `values` is `null`.

```
public TimeUnit getSourceTimeUnit()
```

Returns the time unit in which the values coming from the probability distribution are expressed. If the source unit is `null`, no conversion of the generated values is performed. By default, this returns `null`.

Returns the source time unit.

```
public void setSourceTimeUnit (TimeUnit unit)
```

Sets the source time unit to `unit`.

Parameter

`unit` the source time unit.

See also `getSourceTimeUnit()`

```
public TimeUnit getTargetTimeUnit()
```

Returns the time unit in which the values returned by `nextDouble()` must be expressed. If the target unit is `null`, no conversion of the generated values is performed. By default, this returns `null`.

Returns the target time unit.

```
public void setTargetTimeUnit (TimeUnit unit)
```

Sets the target time unit to `unit`.

Parameter

`unit` the target time unit.

See also `getTargetTimeUnit()`

```
public static double getMean (RandomVariateGen rvg)
```

Returns the mean of the distribution for a random variate generator, taking the shift into account. This method first calls `Distribution.getMean()` on the distribution associated with the generator. If `rvg` is an instance of `RandomVariateGenWithShift` or `RandomVariateGenIntWithShift`, it then subtracts the associated shift.

Parameter

`rvg` the random variate generator.

Returns the possibly shifted mean.

```
public double getMean (int p)
```

Returns the mean for period `p`.

Parameter

`p` the index of the period.

Returns the mean.

```
public double getVariance (int p)
```

Returns the variance for the period `p`.

Parameter

`p` the index of the period.

Returns the variance.

```
public double getMult()
```

Returns the multiplier applied to each generated random variate. The default multiplier is 1.

Returns the applied multiplier.

```
public void setMult (double mult)
```

Sets the multiplier applied to each generated random variate to `mult`.

Parameter

`mult` the new multiplier.

```
public RandomStream getStream()
```

Returns the random stream used during the current period.

```
public Distribution getDistribution()
```

Returns the distribution used during the current period.

```
public double nextDouble (Contact contact)
```

Ignores the given `contact` and calls `nextDouble()`.

```
public static MultiPeriodGen createConstant (PeriodChangeEvent pce, double[]  
                                              values)
```

Constructs and returns a multiple-periods random variate generator using the constant distribution with value `values[p]` for period `p` as defined by `pce`.

Parameters

`pce` the period-change event.

`values` the values of the constant.

Returns the constructed multiple-periods generator.

Throws

`IllegalArgumentException` if the length of array is less than the number of periods.

```
public static MultiPeriodGen createExponential (PeriodChangeEvent pce,  
                                                RandomStream stream,  
                                                double[] lambdas)
```

Constructs and returns a multiple-periods random variate generator using the exponential distribution with rate `lambdas[p]` for period `p` as defined by `pce`. The random stream `stream` is used for all the periods.

Parameters

`pce` the period-change event.

`stream` the random stream.

`lambdas` the rates for the exponential variates.

Returns the constructed multiple-periods generator.

Throws

`IllegalArgumentException` if the length of array is less than the number of periods.

```
public static MultiPeriodGen createGamma (PeriodChangeEvent pce,  
                                           RandomStream stream, double[]  
                                           alphas, double[] lambdas)
```

Constructs and returns a multiple-periods random variate generator using the gamma distribution with parameters `alphas[p]` and `lambdas[p]` for period `p` as defined by `pce`. The random stream `stream` is used for all the periods. The underlying gamma generators use acceptance-rejection rather than inversion for efficiency.

Parameters

`pce` the period-change event.

`stream` the random stream.

`alphas` the alpha parameters for the gamma variates.

`lambdas` the lambda parameters for the gamma variates.

Returns the constructed multiple-periods generator.

Throws

`IllegalArgumentException` if the length of the arrays is less than the number of periods, or the two arrays have different lengths.

ValueGenerator

Represents a value generator for random variates used for simulating contact centers. Implementations of this interface are linked to contact center objects and usually uses a random variate generator to obtain some continuous variates. The generator used or some adjustments made to the value can depend on the concerned contact, the object containing this value generator, the simulation time, etc. This interface defines a method similar to `RandomVariateGen.nextDouble()` but taking a `Contact` object as an argument. This way, random values can depend on the particular contact.

```
package umontreal.iro.lecuyer.contactcenters;  
  
public interface ValueGenerator extends Initializable
```

Methods

```
public double nextDouble (Contact contact)
```

Generates and returns a new value for the contact `contact`. If `contact` is `null` and this is not allowed by the implementation, this method should throw a `NullPointerException`.

Parameter

`contact` the contact being concerned.

Returns the generated value.

Throws

`NullPointerException` if `contact` is illegally `null`.

```
public void init()
```

Initializes the generator at the beginning of the simulation.

ConstantValueGenerator

Implements the `ValueGenerator` interface for a constant and possibly non-stationary value. During each period of the simulation, the generated value is constant for each contact type. When a new period begins, the constant value can be changed.

This implementation only takes contact type identifiers (`Contact.getTypeId()`) and current period (`PeriodChangeEvent.getCurrentPeriod()`) into account for generating values.

```
package umontreal.iro.lecuyer.contactcenters;  
  
public class ConstantValueGenerator implements ValueGenerator
```

Constructors

```
public ConstantValueGenerator (int numTypes, double val)
```

Constructs a new constant stationary value generator supporting `numTypes` contact types, and with value `val` for each contact type.

Parameters

`numTypes` the number of supported contact types.

`val` the value that will be returned by `nextDouble (Contact)`.

```
public ConstantValueGenerator (double[] vals)
```

Constructs a new constant stationary value generator with value `vals[k]` for contact type `k`.

Parameter

`vals` the values for each contact type.

```
public ConstantValueGenerator (PeriodChangeEvent pce, int numTypes, double[]  
                                vals)
```

Constructs a new constant value generator with period-change event `pce`, value `vals[p]` for period `p`, and supporting `numTypes` contact types.

Parameters

`pce` the associated period-change event.

`numTypes` the number of supported contact types.

`vals` the generated value for each period.

Throws

`IllegalArgumentException` if a value is not specified for each period.

```
public ConstantValueGenerator (PeriodChangeEvent pce, double[] [] vals)
```

Constructs a new constant value generator with values `vals` and period-change event `pce`. The array element `vals[p][k]` gives the value for period `p`, contact type `k`.

Parameters

`pce` the associated period-change event.

`vals` the array of values.

Throws

`IllegalArgumentException` if an array of values is not specified for each period.

Methods

```
public double[] [] getValues()
```

Returns the values used by this generator. The format of the array is the same as in the last constructor.

Returns the associated values.

```
public void setValues (double[] [] vals)
```

Sets the values for this generator to `vals`. This method can be used to change the number of supported contact types, but it cannot be used to change the number of periods.

Parameter

`vals` the new values for this generator.

Throws

`IllegalArgumentException` if the length of the given array is incorrect.

```
public double nextDouble (Contact contact)
```

Returns the value of the constant corresponding to the type of `contact`, and the current period. If the contact type identifier is greater than or equal to the number of supported contact types, or a type smaller than zero, an exception is thrown.

RandomValueGenerator

Implements the `ValueGenerator` interface when the values come from a continuous and possibly non-stationary distribution. For each period and contact type, a different random variate generator can be used to get a value. This class can be instantiated the same way a `ConstantValueGenerator` is constructed, replacing constants with random variate generators.

```
package umontreal.iro.lecuyer.contactcenters;
```

```
public class RandomValueGenerator implements ValueGenerator
```

Constructors

```
public RandomValueGenerator (int numTypes, RandomVariateGen gen)
```

Constructs a new random stationary value generator with generator `gen` for each contact type, and supporting `numTypes` contact types.

Parameters

`numTypes` the number of supported contact types.

`gen` the random variate generator used for all contact types.

```
public RandomValueGenerator (RandomVariateGen[] gens)
```

Constructs a new random stationary value generator with generator `gens[k]` for contact type `k`.

Parameter

`gens` the random variate generators used by this object.

```
public RandomValueGenerator (PeriodChangeEvent pce, int numTypes,  
                             RandomVariateGen[] gens)
```

Constructs a new random value generator with period-change event `pce`, generator `gens[p]` for period `p`, and supporting `numTypes` contact types.

Parameters

`pce` the associated period-change event.

`numTypes` the number of supported contact types.

`gens` the array containing a generator for each period.

Throws

`IllegalArgumentException` if a generator is not specified for each period.

```
public RandomValueGenerator (PeriodChangeEvent pce, RandomVariateGen[] []  
                             gens)
```

Constructs a new random value generator with period-change event `pce` and random variate generators `gens`. For the period `p` and contact type `k`, the random variate generator `gens[p][k]` is used.

Parameters

`pce` the associated period-change event.

`gens` the array of generators for each period and contact type.

Throws

`IllegalArgumentException` if an array of generators is not specified for each period.

Methods

```
public RandomVariateGen[] [] getRandomVariateGens()
```

Returns the array of random variate generators associated with this object. The format of this array is the same as the array passed to the last constructor.

Returns the random variate generators for this object.

```
public void setRandomVariateGens (RandomVariateGen[] [] gens)
```

Sets the random variate generators for this object to `gens`. This method can be used to change the number of supported contact types, but it cannot be used to change the number of periods.

Parameter

`gens` the new random variate generators for this object.

Throws

`IllegalArgumentException` if the length of the given array is incorrect.

MinValueGenerator

Value generator for the minimum of values. This value generator defines an internal array of value generators used by the `nextDouble (Contact)` method. When `nextDouble (Contact)` is called, a value is generated using all registered value generators and the minimum value, generated by $v[j^*]$, is returned. It is also possible to get the index j^* of the value generator having returned the minimum as well as all the generated values. This class is used by waiting queues and agent groups to generate the maximal queue times and service times.

```
package umontreal.iro.lecuyer.contactcenters;
```

```
public class MinValueGenerator implements ValueGenerator
```

Constructors

```
public MinValueGenerator()
```

Constructs a minimum value generator with no registered internal value generator.

```
public MinValueGenerator (int initialLength)
```

Constructs a minimum value generator with an internal array of value generators containing `initialLength` elements.

Parameter

`initialLength` the number of elements in the internal array.

Methods

```
public void compact()
```

Recreates the internal array of value generators for its length to correspond to `getMaxVType()`.

```
public int getMaxVType()
```

Returns the maximum index (non-inclusive) for which `getValueGenerator (int)` returns a non-null value.

Returns the highest index of registered value generators.

```
public ValueGenerator getValueGenerator (int vType)
```

Returns the value generator corresponding to value type `vType`.

Parameter

`vType` the queried value type.

Returns the associated value generator.

```
public void setValueGenerator (int vType, ValueGenerator vgen)
```

Sets the value generator corresponding to value type `vType` to `vgen`.

Since the value generators are stored in an internal array, it is recommended to use small value types. This will avoid the creation of large arrays of `null`'s.

Parameters

`vType` the affected value type.

`vgen` the new value generator.

```
public boolean isKeepingValues()
```

Determines if the value generator is keeping all the generated values used to compute the last minimum.

Returns the generated values keeping indicator.

```
public void setKeepingValues (boolean k)
```

Sets the keeping-values indicator to `k`.

Parameter

`k` the new keeping-values indicator.

```
public void init()
```

Initializes all the associated value generators.

```
public double nextDouble (Contact contact)
```

Generates and returns a new value for the contact `contact`. For each associated value generator, this method calls `nextDouble` and returns the minimal value. If there is no associated value generator, `Double.NaN` is returned.

Parameter

`contact` the contact for which a value is generated.

Returns the generated value.

```
public double getLastValue()
```

Returns the last value returned by `nextDouble (Contact)`.

Returns the last generated value.

```
public int getLastVType()
```

Returns the value type for the last value. This corresponds to the index of the value generator, as returned by `getValueGenerator (int)`, having generated the chosen minimal value.

Returns the index of the minimal value.

```
public double[] getLastValues()
```

Returns all the generated values upon the last call to `nextDouble (Contact)` if `isKeepingValues()` returns `true`, or throws an `IllegalStateException`. The length of the returned array corresponds to `getMaxVType()`.

Returns the generated values.

Throws

`IllegalStateException` if the object does not keep generated values.

```
public double getLastValue (int vType)
```

Returns the value generated by `getValueGenerator (vType)` upon the last call to `nextDouble (Contact)` if `isKeepingValues()` returns `true`, or throws an `IllegalStateException`. If the associated value generator is `null`, `Double.NaN` is returned.

Parameter

`vType` the queried value type.

Returns the generated value.

Throws

`IllegalStateException` if the object does not keep generated values.

`ArrayIndexOutOfBoundsException` if `vType` is negative or greater than or equal to `getMaxVType()`.

ContactCenter

Defines utility methods for contact center simulation. This class provides facilities to initialize the contact center's objects, and to perform some actions on a group of objects.

```
package umontreal.iro.lecuyer.contactcenters;
```

```
public class ContactCenter
```

Methods

```
public static void initElements (Iterable<?> el)
```

Initializes all elements enumerated by the iterable `el`. This method calls the `init` method of any iterated object which is an instance of `Initializable`, `MeasureMatrix`, `ListOfStatProbes`, and `MatrixOfStatProbes`. For other elements instance of `Iterable`, this method is called recursively.

```
public static void initElements (Object[] el)
```

Equivalent to `initElements (Iterable)` for an array of objects.

Parameter

`el` the array of elements.

```
public static void initElements (Initializable[] el)
```

Initializes all elements in `el`. For each `Initializable` object in the array, calls the `Initializable.init()` method.

```
public static void initElements (MeasureMatrix[] el)
```

Initializes all elements in `el`. For each `MeasureMatrix` object in the array, calls the `MeasureMatrix.init()` method.

```
public static void initElements (StatProbe[] el)
```

Initializes all elements in `el`. For each `StatProbe` object in the array, calls the `StatProbe.init()` method.

```
public static void initElements (ListOfStatProbes<?>[] el)
```

Initializes all elements in `el`. For each `ListOfStatProbes` object in the array, calls the `ListOfStatProbes.init()` method.

```
public static void initElements (MatrixOfStatProbes<?>[] el)
```

Initializes all elements in `el`. For each `MatrixOfStatProbes` object in the array, calls the `MatrixOfStatProbes.init()` method.

```
public static void toggleElements (Iterable<? extends ToggleElement> el,
                                   boolean enabled)
```

Toggles the elements to the status `enabled`. For each `ToggleElement` object enumerated by the iterable `el`, calls the `ToggleElement.start()` or `ToggleElement.stop()` methods.

Parameters

`e1` the list of toggle elements.

`enabled` `true` if the toggle elements are enabled, `false` if they are disabled.

```
public static void toggleElements (ToggleElement[] e1, boolean enabled)
```

Toggles the elements to the status `enabled`. For each `ToggleElement` object in the array `e1`, calls the `ToggleElement.start()` or `ToggleElement.stop()` methods.

Parameters

`e1` the array of toggle elements.

`enabled` `true` if the toggle elements are enabled, `false` if they are disabled.

```
public static void startPeriodChangeEvents (Iterable<? extends  
                                             PeriodChangeEvent> pce)
```

For each period-change event enumerated by the iterable `pce`, calls the `PeriodChangeEvent.start()` method.

Parameter

`pce` the list of period-change events.

```
public static void startPeriodChangeEvents (PeriodChangeEvent[] pce)
```

For each period-change event in the array `pce`, calls the `PeriodChangeEvent.start()` method.

Parameter

`pce` the array of period-change events.

```
public static void stopPeriodChangeEvents (Iterable<? extends  
                                           PeriodChangeEvent> pce)
```

For each period-change event enumerated by the iterable `pce`, calls the `PeriodChangeEvent.stop()` method.

Parameter

`pce` the list of period-change events.

```
public static void stopPeriodChangeEvents (PeriodChangeEvent[] pce)
```

For each period-change event in the array `pce`, calls the `PeriodChangeEvent.stop()` method.

Parameter

`pce` the array of period-change events.

```
public static void clearWaitingQueues (Iterable<? extends WaitingQueue>  
                                       waitingQueues, int dqType)
```

Clears all waiting queues enumerated by the iterable `waitingQueues` with dequeue type `dqType`. For each `WaitingQueue` object in the list, calls the `WaitingQueue.clear()` method with the given `dqType`.

Parameters

`waitingQueues` the list of waiting queues.

`dqType` the dequeue type being used.

Throws

`NullPointerException` if the given list is `null`.

```
public static void clearWaitingQueues (WaitingQueue[] waitingQueues, int
                                     dqType)
```

Clears all waiting queues in `waitingQueues` with dequeue type `dqType`. For each `WaitingQueue` object in the array, calls the `WaitingQueue.clear()` method with the given `dqType`.

Parameters

`waitingQueues` the array of waiting queues.

`dqType` the dequeue type being used.

Throws

`NullPointerException` if the given list is `null`.

```
public static void clearWaitingQueues (WaitingQueueSet[] waitingQueues,
                                     int dqType)
```

Clears all waiting queues in `waitingQueues` with dequeue type `dqType`. For each `WaitingQueueSet` object in the array, clears all registered waiting queues with the given `dqType`.

Parameters

`waitingQueues` the array of waiting queues.

`dqType` the dequeue type being used.

Throws

`NullPointerException` if the given list is `null`.

```
public static String toShortString (Named named)
```

Returns a short string representation of the named object `named`. If the length of `named.getName()` is greater than 0, returns that name. Otherwise, this returns the result of the `toString` method defined in `Object`.

Returns a short string representation of the named object.

MatrixUtil

Contains utility methods to add rows or columns to matrices, and to construct a matrix by repeating a submatrix several times.

```
package umontreal.iro.lecuyer.contactcenters;
```

```
public class MatrixUtil
```

Methods

```
public static DoubleMatrix2D getCost (DoubleMatrix2D m, double[] cost)
```

Converts the matrix `m` into a matrix of costs using the cost vector `cost`. The matrix `m` should contain counts of events, e.g., the number of arrivals, or the integral over simulation time of a quantity, e.g., the queue size. Each row corresponds to one count and each column represents one period. Assuming the cost vector is a row vector in \mathbb{R}^d , the method computes $C * M$, and stores the result in the matrix `m`. C is a $d \times d$ matrix with the costs on its diagonal, i.e., $C_{k,k} = \text{costs}[k]$, and $C_{i,j} = 0$ for $i \neq j$. M is a $d \times P$ matrix stored in `m`. If `m` has $d + 1$ rows, the last row of the matrix is filled with the total costs, i.e.,

$$M_{d,j} = \sum_{i=0}^{d-1} M_{i,j} C_{i,i}$$

for $j = 0, \dots, P - 1$.

Parameters

`m` the matrix of values.

`cost` the cost vector.

Returns the given matrix `m`.

Throws

`IllegalArgumentException` if the length of `cost` does not correspond to `m.rows()` or `m.rows() - 1`.

```
public static DoubleMatrix2D addSumRow (DoubleMatrix2D m)
```

Equivalent to `addSumRow (m, false)`.

Parameter

`m` the matrix being processed.

Returns the matrix with the added row of sums.

```
public static DoubleMatrix2D addSumRow (DoubleMatrix2D m, boolean always)
```

Makes a copy of the matrix `m` with a new row containing the sum of each column. If `m` has a single row and if `always` is set to `false`, the matrix is returned unchanged. Otherwise, a new matrix is created with the additional row of sums. The sums of columns are stored in the last row of the returned matrix.

Parameters

m the matrix being processed.

always if **true**, the row is added even if **m** has one row.

Returns the matrix with the added row of sums.

```
public static DoubleMatrix2D addSumColumn (DoubleMatrix2D m)
```

Equivalent to `addSumColumn (m, false)`.

Parameter

m the matrix being processed.

Returns the matrix with the added column of sums.

```
public static DoubleMatrix2D addSumColumn (DoubleMatrix2D m, boolean  
                                           always)
```

This method, similar to `addSumRow (DoubleMatrix2D, boolean)`, adds an extra column to the matrix **m** for the sum of each column.

Parameters

m the matrix being processed.

always determines if the column is always added.

Returns the matrix with the added column of sums.

```
public static DoubleMatrix2D repMat (DoubleMatrix2D m, int numRows, int  
                                   numCols)
```

Constructs a matrix by copying **m** a certain number of times. The new matrix contains **numRows*numCols** copies of **m** tiled in a grid with dimensions **numRows**×**numCols**. If **numRows** and **numCols** are both 1, the matrix is returned unchanged.

Parameters

m the matrix to be tiled.

numRows the number of rows containing copies of **m**.

numCols the number of columns containing copies of **m**.

Returns the matrix containing copies of **m**.

RandomStreamUtil

Provides utility methods to create and extend arrays of random streams.

```
package umontreal.iro.lecuyer.contactcenters;
```

```
public class RandomStreamUtil
```

Methods

```
public static RandomStream[] createRandomStreamArray (RandomStream[]
                                                    oldArray, int size,
                                                    RandomStreamFactory
                                                    rsf)
```

Creates or extends an array of random streams. This can be useful when reconstructing a contact center with new parameters, to keep as many random streams as possible for maximizing random number synchronization.

If `oldArray` is `null`, a new array of length `size` will be allocated and filled with random streams. If `oldArray` is not `null` and its length is greater than or equal to `size`, it is returned unchanged. Otherwise, a new array is created, the already constructed random streams are copied and new ones are constructed to fill the array. The random streams are created using the given random stream factory. The method returns an array of random streams with length greater than or equal to `size`.

Parameters

`oldArray` the old array of random streams.

`size` the minimal size of the returned array.

`rsf` the random stream factory used to create the random streams.

Returns the constructed array of random streams.

```
public static RandomStream[][] createRandomStreamMatrix
(RandomStream[][] oldMatrix, int rows, int columns, RandomStreamFactory
rsf)
```

Creates or extends a matrix (i.e., 2D array) of random streams. This can be useful when reconstructing a contact center with new parameters, to keep as many random streams as possible for maximizing random number synchronization.

If `oldMatrix` is `null`, a new array of length `rows` will be allocated and filled with arrays of random streams. If `oldMatrix` is not `null` and its length is greater than or equal to `rows`, it is returned unchanged. Otherwise, a new array is created, the already constructed arrays of random streams are copied and new ones are constructed to fill the array. The internal arrays of random streams are created using `createRandomStreamArray (RandomStream[], int, RandomStreamFactory)`.

Parameters

`oldMatrix` the old matrix of random streams.

`rows` the required number of rows.

`columns` the required number of columns.

`rsf` the random stream factory used to create streams.

Returns the new matrix of random streams.

StatUtil

Provides methods to add ratios into lists and matrices of statistical probes as well as a method to trim arrays of observations in statistical probes in order to save memory.

```
package umontreal.iro.lecuyer.contactcenters;
```

```
public class StatUtil
```

Methods

```
public static void addRatio (MatrixOfTallies<?> mt, DoubleMatrix2D x,
                             DoubleMatrix2D y)
```

Equivalent to `add (mt, x, y, 1.0, Double.NaN)`.

```
public static void addRatio (MatrixOfTallies<?> mt, DoubleMatrix2D x,
                             DoubleMatrix2D y, double mult)
```

Equivalent to `add (mt, x, y, mult, Double.NaN)`.

```
public static void addRatio (MatrixOfTallies<?> mt, DoubleMatrix2D x,
                             DoubleMatrix2D y, double mult, double
                             zeroOverZero)
```

For each tally (r, c) in the matrix of tallies `mt`, adds the ratio `mult*x.get (r, c)/y.get (r, c)`. If a 0/0 ratio must be added, the value `zeroOverZero` is used instead of `Double.NaN`.

Parameters

`mt` the target matrix of tallies.

`x` the numerator matrix.

`y` the denominator matrix.

`mult` the multiplier of the ratio.

`zeroOverZero` the value for 0/0.

Throws

`IllegalArgumentException` if the dimensions of `x` or `y` do not correspond to the dimensions of the matrix of tallies.

```
public static void addRatio (ListOfTallies<?> mt, double[] x, double[] y)
    Equivalent to add (mt, x, y, 1.0).
```

```
public static void addRatio (ListOfTallies<?> at, double[] x, double[] y,
                             double mult)
```

For each tally i in the list of tallies `at`, adds the ratio `mult*x[i]/y[i]`.

Parameters

- at** the target list of tallies.
- x** the numerator array.
- y** the denominator array.
- mult** the multiplier of the ratio.

Throws

- IllegalArgumentException** if the length of **x** or **y** do not correspond to the length of the list of tallies.

```
public static MatrixOfFunctionOfMultipleMeansTallies<
FunctionOfMultipleMeansTally> createMatrixOfRatioTallies
(MatrixOfTallies<?> upper, MatrixOfTallies<?> lower)
```

Creates a matrix of ratio tallies from two matrices of tallies. This method takes two matrices which must have the same dimensions. The ratio tally for row *r* and column *c* is constructed by taking the tallies at corresponding row and column in **upper** and **lower**.

Parameters

- upper** the matrix of upper parts of ratios.
- lower** the matrix of lower parts of ratios.

Returns the new matrix of ratio tallies.

```
public static void compactProbes (Iterable<?> probes)
```

Trims the internal arrays of statistical probes listed in **probes** to minimize memory utilization. For each **TallyStore** object in **probes** or in an array or matrix added to **probes**, calls **TallyStore.getArray().trimToSize()**.

Parameter

- probes** the list of statistical probes to process.

RepSimCC

Extends `RepSim` to use measure matrices as counters, to compute observations. This class defines a list matrices of measures that can be added to. At the beginning of each replication, the matrices are initialized, and the program updates them. Each column of such matrices usually corresponds to a period as defined by a period-change event. At the end of the replication, in the `RepSim.addReplicationObs (int)` method, values are extracted from the matrices before they are added to matrices of tallies.

```
package umontreal.iro.lecuyer.contactcenters;

public abstract class RepSimCC extends RepSim
```

Constructors

```
public RepSimCC (int minReps)
    Calls super (minReps).

public RepSimCC (Simulator sim, int minReps)
    Calls RepSim.RepSim (Simulator, int).

public RepSimCC (int minReps, int maxReps)
    Calls super (minReps, maxReps).

public RepSimCC (Simulator sim, int minReps, int maxReps)
    Calls RepSim.RepSim (Simulator, int, int).
```

Methods

```
public List<MeasureMatrix> getMeasureMatrices()
    Returns the matrices of measures registered to this object. These matrices must be capable
    of supporting multiple periods. The returned list should contain non-null instances of
    MeasureMatrix only.
    Returns the list of measure matrices.

public void performReplication (int r)
    This method is overridden to initialize the matrices of measures after the simulator is ini-
    tialized.

public static DoubleMatrix2D getReplicationValues (MeasureMatrix mat,
                                                    DoubleMatrix2D m)
    Computes the matrix of observations from the matrix of measures mat, and stores the result
    in m. The returned matrix has the same number of rows as the number of measures and the
    same number of columns as the number of periods. Element (r, c) of the matrix is given
    by mat.getMeasure (r, c).
```

Parameters

mat the matrix of measures for which observations are queried.

m the matrix of `double`'s filled with the result.

Returns the given matrix **m**.

Throws

`IllegalArgumentException` if the dimensions of the matrix are invalid.

`NullPointerException` if **mat** or **m** are null.

```
public static DoubleMatrix2D getReplicationValues (MeasureMatrix mat)
```

Constructs a matrix **m** with as many rows as the number of measures in **mat** and as many columns as the number of periods, calls `getReplicationValues (mat, m)` to fill the matrix, and returns it.

Parameter

mat the measure matrix for which observations are queried.

Returns the matrix filled with the result.

Throws

`NullPointerException` if **mat** is null.

```
public static DoubleMatrix2D getReplicationValues (MeasureMatrix mat,
                                                    DoubleMatrix2D m,
                                                    boolean preliminary,
                                                    boolean wrapup, boolean[]
                                                    mainPeriods)
```

Computes the matrix of observations for the measure matrix **mat** and stores the result in **m**. It is assumed that the matrix contains observations for **np** periods, including a preliminary and a wrap-up periods. The matrix **m** must contain observations for main periods only as well as the time-aggregate observations. If **preliminary** is set to `true`, the observations of the preliminary period will be included in the time-aggregate count. If **wrapup** is `true`, the observations in the wrap-up period will be included. If **mainPeriods** is null, all main periods will be included in the aggregate values. Otherwise, the value for (main) period **p** will be included in the aggregated sum if and only if **mainPeriods**[**p** - 1] is `true`.

Each column of the matrix corresponds to one period and the last column contains the values for the whole replication. Each row corresponds to one type of measure.

Parameters

mat the measure matrix for which observations are queried.

m the matrix filled with the result.

preliminary if the preliminary period is included in the last column of the matrix.

wrapup if the wrap-up period is included in the last column of the matrix.

mainPeriods indicates which main periods are included in the aggregate measure.

Returns the given matrix `m`.

Throws

`IllegalArgumentException` if the dimensions of the matrix are invalid, or if `mainPeriods` is non-null and has an invalid length.

`NullPointerException` if `mat` or `m` are null.

```
public static DoubleMatrix2D getReplicationValues (MeasureMatrix mat,  
                                                  DoubleMatrix2D m,  
                                                  boolean preliminary,  
                                                  boolean wrapup)
```

Equivalent to `getReplicationValues (mat, m, preliminary, wrapup, null)`.

Parameters

`mat` the measure matrix for which observations are queried.

`m` the matrix filled with the result.

`preliminary` if the preliminary period is included in the last column of the matrix.

`wrapup` if the wrap-up period is included in the last column of the matrix.

Returns the given matrix `m`.

Throws

`IllegalArgumentException` if the dimensions of the matrix are invalid.

`NullPointerException` if `mat` or `m` are null.

```
public static DoubleMatrix2D getReplicationValues (MeasureMatrix mat,  
                                                  boolean preliminary,  
                                                  boolean wrapup, boolean[]  
                                                  mainPeriods)
```

Computes the matrix of observations for the measure matrix `mat`, and returns the result in a matrix. This method uses `getReplicationValues` for the computation.

Parameters

`mat` the measure matrix for which observations are queried.

`preliminary` if the preliminary period is included in the last column of the matrix.

`wrapup` if the wrap-up period is included in the last column of the matrix.

`mainPeriods` indicates which main periods are included in the aggregated measure.

Returns the matrix filled with the result.

Throws

`NullPointerException` if `mat` or `m` are `null`.

`IllegalArgumentException` if `mainPeriods` is not `null` and has an invalid length.

```
public static DoubleMatrix2D getReplicationValues (MeasureMatrix mat,  
                                                  boolean preliminary,  
                                                  boolean wrapup)
```

Equivalent to `getReplicationValues (mat, preliminary, wrapup, null)`.

Parameters

`mat` the measure matrix for which observations are queried.

`preliminary` if the preliminary period is included in the last column of the matrix.

`wrapup` if the wrap-up period is included in the last column of the matrix.

Returns the matrix filled with the result.

Throws

`NullPointerException` if `mat` or `m` are `null`.

```
public static DoubleMatrix2D timeNormalize (PeriodChangeEvent pce,  
                                           DoubleMatrix2D m, double  
                                           totalTime)
```

Normalizes the matrix `m` using simulation time. Usually, this method receives a matrix produced by `getReplicationValues (MeasureMatrix, DoubleMatrix2D)`. It assumes that each row corresponds to a count or an integral and one column corresponds to a main period. If there is one more column than the number of main periods, the last column corresponds to values for the whole replication. Each element of the matrix is divided by a simulation time determined by the period-change event `pce`. For each column `c` corresponding to one main period, each row is divided by the period duration obtained using `PeriodChangeEvent.getPeriodDuration (c + 1)`. For the column corresponding to the whole replication, the rows are divided by the total simulation time `totalTime`.

Parameters

`pce` the period-change event defining the periods.

`m` the matrix being normalized.

`totalTime` the supplied total simulation time.

Returns the given matrix `m`.

Throws

`IllegalArgumentException` if the number of columns of `m` is incorrect.

```
public static DoubleMatrix2D timeNormalize (PeriodChangeEvent pce,  
                                           DoubleMatrix2D m, boolean  
                                           preliminary, boolean wrapup,  
                                           boolean[] mainPeriods)
```

Equivalent to `timeNormalize (PeriodChangeEvent, DoubleMatrix2D, double)` with automatic computation of total simulation time. The total time is computed by summing the duration of the periods defined by `pce`. The parameters `preliminary`, `wrapup` and `mainPeriods` play the same role as with `getReplicationValues (MeasureMatrix, DoubleMatrix2D)`.

Parameters

`pce` the period-change event defining the periods.

`m` the matrix being normalized.

`preliminary` determines if the preliminary period is included in the time-aggregate values.

`wrapup` determines if the wrap-up period is included in the time-aggregate values.

`mainPeriods` indicates which main periods are included in the aggregated measure.

Returns the given matrix `m`.

Throws

`IllegalArgumentException` if the number of columns of `m` is incorrect.

```
public static DoubleMatrix2D timeNormalize (PeriodChangeEvent pce,  
                                           DoubleMatrix2D m, boolean  
                                           preliminary, boolean wrapup)
```

Equivalent to `timeNormalize (pce, m, preliminary, wrapup, null)`.

Parameters

`pce` the period change event defining the periods.

`m` the matrix being normalized.

`preliminary` determines if the preliminary period is included in the time-aggregate values.

`wrapup` determines if the wrap-up period is included in the time-aggregate values.

Returns the given matrix `m`.

Throws

`IllegalArgumentException` if the number of columns of `m` is incorrect.

BatchMeansSimCC

Extends `BatchMeansSim` to use matrices of measures for storing the intermediate values \mathbf{V}_j 's of real batches. This class defines a list of measure matrices which is automatically initialized after the warmup period. When batch aggregation is turned OFF, these matrices contain a single period and are reinitialized at the beginning of each batch. If batch aggregation is turned ON, each period in measure matrices correspond to a real batch.

```
package umontreal.iro.lecuyer.contactcenters;
```

```
public abstract class BatchMeansSimCC extends BatchMeansSim
```

Constructors

```
public BatchMeansSimCC (int minBatches, double batchSize, double
                        warmupTime)
```

Calls `super (minBatches, batchSize, warmupTime)`.

```
public BatchMeansSimCC (Simulator sim, int minBatches, double batchSize,
                        double warmupTime)
```

Calls `BatchMeansSim.BatchMeansSim (Simulator, int, double, double)`.

```
public BatchMeansSimCC (int minBatches, int maxBatches, double batchSize,
                        double warmupTime)
```

Calls `super (minBatches, maxBatches, batchSize, warmupTime)`.

```
public BatchMeansSimCC (Simulator sim, int minBatches, int maxBatches,
                        double batchSize, double warmupTime)
```

Calls `BatchMeansSim.BatchMeansSim (Simulator, int, int, double, double)`.

Methods

```
public List<MeasureMatrix> getMeasureMatrices()
```

Returns the matrices of measures registered to this object. These matrices must be capable of supporting multiple periods. The returned list should contain non-null instances of `MeasureMatrix` only.

Returns the list of measure matrices.

```
public void initBatchStat()
```

Initializes registered matrices of measures if batch aggregation is turned OFF.

```
public void initRealBatchProbes()
```

Initializes the matrices of measures.

```
public void addRealBatchObs()
```

For each `IntegralMeasureMatrix` instance in the list returned by `getMeasureMatrices()`, calls `IntegralMeasureMatrix.newRecord()`.

```
public static DoubleMatrix2D getBatchValues (MeasureMatrix mat,
                                             DoubleMatrix2D m, int s, int
                                             h)
```

Copies the values corresponding to the current effective batch for the measure matrix `mat` into the matrix `m`. The given matrix of measures should be registered to this object for this method to be used. The Colt matrix `m` must have one row for each measure, and a single column. The method returns `m` after it is filled.

Parameters

`mat` the measure matrix for which the Colt matrix is required.

`m` the matrix receiving the results.

`s` the starting real batch.

`h` the number of real batches per effective batch.

Throws

`IllegalArgumentException` if the dimensions of the matrix `m` are incompatible.

```
public static DoubleMatrix2D getBatchValues2D (MeasureMatrix mat, int s,
                                             int h)
```

Constructs a matrix with one row for each measure in `mat` and a single column, then calls `getBatchValues (MeasureMatrix, DoubleMatrix2D, int, int)`.

```
public static DoubleMatrix1D getBatchValues (MeasureMatrix mat,
                                             DoubleMatrix1D m, int s, int
                                             h)
```

Equivalent to `getBatchValues (MeasureMatrix, DoubleMatrix2D, int, int)` for a `DoubleMatrix1D` instance.

```
public static DoubleMatrix1D getBatchValues1D (MeasureMatrix mat, int s,
                                             int h)
```

Constructs a matrix with one row for each measure in `mat`, and calls `getBatchValues (MeasureMatrix, DoubleMatrix1D, int, int)`.

```
public static double[] getBatchValues (MeasureMatrix mat, double[] m, int  
                                     s, int h)
```

Equivalent to `getBatchValues (MeasureMatrix, DoubleMatrix2D, int, int)` for an array.

```
public static double[] getBatchValues (MeasureMatrix mat, int s, int h)
```

Constructs an array with one element for each measure in `mat`, and calls `getBatchValues (MeasureMatrix, double[], int, int)`.

```
public static DoubleMatrix2D timeNormalize (DoubleMatrix2D m, double l)
```

Normalizes the rows of the one-column matrix `m` using the batch length `l`. Each row of the matrix is divided by `l` and the modified matrix is returned.

Parameters

`m` the matrix being normalized.

`l` the batch length.

Returns the modified matrix.

```
public static DoubleMatrix1D timeNormalize (DoubleMatrix1D m, double l)
```

Equivalent to `timeNormalize (DoubleMatrix2D, double)` with an instance of `DoubleMatrix1D`.

```
public static double[] timeNormalize (double[] m, double l)
```

Equivalent to `timeNormalize (DoubleMatrix2D, double)` for an array.

Package `umontreal.iro.lecuyer.contactcenters.contact`

Manages contact arrivals into the contact center system. Any *contact* traveling in the system is represented by an object from the `Contact` class being defined in this package. This class defines a number of attributes associated with all contacts, and the user can add custom attributes by defining a subclass.

Although an object representing a contact can be freely instantiated, it is usually constructed by a *contact source*. Two types of contact sources are available: *contact arrival processes* provided by this package, and *dialers* supported by the `umontreal.iro.lecuyer.contactcenters.dialer` package. Arrival processes determine when contact objects need to be created, according to specific (stochastic) arrival processes. Each concrete arrival process must correspond to an algorithm for generating inter-arrival times. These times could depend on the entire state of the system in a complicated way, but they often depend only on the simulation time and previous inter-arrival times. For each process, the first arrival is scheduled when the arrival process is started, often at the beginning of the simulation. Figure 1 gives a UML diagram for this contact creation facility.

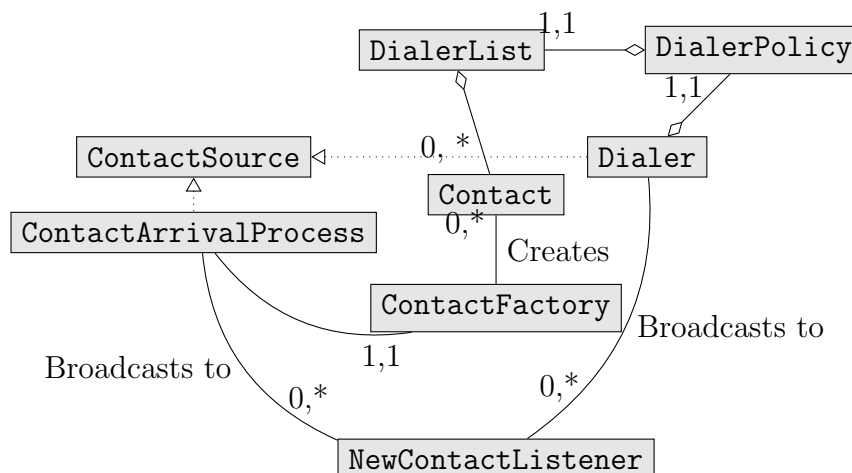


Figure 1: UML diagram describing the facilities for creating contacts

The *factory* design pattern is used to allow the sources to construct contacts without knowing their types explicitly. The `ContactFactory` interface specifies a method called `newInstance` returning a newly-constructed and configured contact object. A contact source can create contacts from any class that implements this interface simply by invoking this `newInstance` method. Thus, changing the type of contact (and the name of its explicit constructor) requires no change to the implementation of the contact source.

When a new contact occurs, it is instantiated by the associated factory and broadcast to the registered *new-contact listeners*. Then the next arrival is scheduled. Each contact source is assigned a factory that typically constructs contacts of a single type. All contact sources can be initialized, started, and stopped.

Contact

Represents a contact (phone call, fax, e-mail, etc.) into the contact center. A contact enters the system at a given time, and requires some form of service. If it cannot be served immediately, it joins a queue or leaves the system. In more complex contact centers, contacts can be served more than once and can join several queues sequentially. A contact object holds all the information about a single contact. The arrival time, the total time spent in queue, the total time spent in service, the last joined queue, and the last serving agent group can be obtained from any contact object. Information about the complete path of the contact into the system can also be stored, but this is disabled by default to reduce memory usage.

For easier indexing in skill-based routers, every contact has a numerical type identifier. For waiting queues supporting it, a contact object also holds a priority. The **Contact** class implements the **Comparable** interface which allows to define the default priorities when contacts are in priority queues.

Extra information can be added to a contact object using two different mechanisms: by adding attributes to the map returned by `getAttributes()`, or by defining fields in a subclass. The `getAttributes()` method returns a **Map** that can be used to define custom contact attributes dynamically. This can be used for quick implementation of user-defined attributes, but it can reduce performance of the application since look-ups in a map are slower than direct manipulation of fields. Alternatively, this class can be extended to add new attributes as fields. However, the contact subclass will have to be used for communication between parts of the program needing the extra information, which involves casts.

By default, no trunk group is associated with contacts. As a result, every contact can enter the system, since its capacity is infinite. If a contact is associated with a trunk group using `setTrunkGroup (TrunkGroup)`, a line is allocated by the router at the time of its arrival. If no line is available in the associated trunk group, the contact is blocked. Otherwise, it is processed and the channel is released when it exits.

Each contact has an associated simulator which is used to schedule contact-related events such as abandonment or service termination. This simulator is determined at the time of construction. If a constructor accepting a **Simulator** instance is called, the given simulator is used. Otherwise, the default simulator returned by `Simulator.getDefaultSimulator()` is used.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class Contact implements Comparable<Contact>, Named, Cloneable
```

Constructors

```
public Contact()
```

Constructs a new contact object with type identifier 0, priority 1, and the default simulator.

```
public Contact (Simulator sim)
```

Equivalent to constructor `Contact()`, with the given simulator `sim`.

Parameter

`sim` the simulator attached to the new contact.

Throws

`NullPointerException` if `sim` is `null`.

```
public Contact (int typeId)
```

Constructs a new contact with priority 1, type identifier `typeId`, and the default simulator.

Parameter

`typeId` type identifier of the new contact.

Throws

`IllegalArgumentException` if the type identifier is negative.

```
public Contact (Simulator sim, int typeId)
```

Equivalent to constructor `Contact (int)`, with the given simulator `sim`.

Parameters

`sim` the simulator attached to the new contact.

`typeId` type identifier of the new contact.

Throws

`NullPointerException` if `sim` is `null`.

`IllegalArgumentException` if the type identifier is negative.

```
public Contact (double priority, int typeId)
```

Constructs a new contact object with a priority `priority`, type identifier `typeId`, and the default simulator. The contact type identifier must be non-negative while the priority can be any value. The smaller is the value of `priority`, the greater is the priority of the contact.

Parameters

`priority` the contact's priority.

`typeId` the type identifier of this contact.

Throws

`IllegalArgumentException` if the type identifier is negative.

```
public Contact (Simulator sim, double priority, int typeId)
```

Equivalent to constructor `Contact (double, int)`, with the given simulator `sim`.

Parameters

`sim` the simulator attached to the new contact.

`priority` the contact's priority.

`typeId` type identifier of the new contact.

Throws

`NullPointerException` if `sim` is null.

`IllegalArgumentException` if the type identifier is negative.

Methods

```
public final Simulator simulator()
```

Returns a reference to the simulator attached to this contact.

Returns the simulator attached to this contact.

```
public final void setSimulator (Simulator sim)
```

Sets the simulator attached to this contact to `sim`. This method should not be called while this contact is in a waiting queue, or being served. The main use of this method is for splitting: a contact is cloned, and a new simulator is assigned to the copy while the original contact keeps the old simulator.

Parameter

`sim` the new simulator.

Throws

`NullPointerException` if `sim` is null.

```
public double getArrivalTime()
```

Returns the contact's arrival simulation time. This is the simulation time at which the contact object was constructed.

Returns the arrival simulation time of this contact.

```
public void setArrivalTime (double arrivalTime)
```

Sets the arrival time of this contact to `arrivalTime`. This method should be called before the contact enters into a waiting queue or an agent group.

Parameter

`arrivalTime` the new arrival time.

Throws

`IllegalArgumentException` if `arrivalTime` is negative.

```
public double getDefaultPatienceTime()
```

Returns the default patience time for this contact object. This corresponds to the maximal queue time before the contact abandons the queue. By default, this is `Double.POSITIVE_INFINITY`, i.e., no abandonment occurs.

Returns the patience time of the contact.

```
public void setDefaultPatienceTime (double patienceTime)
```

Sets the default patience time of this contact to `patienceTime`.

Parameter

`patienceTime` the new patience time of the contact.

Throws

`IllegalArgumentException` if the given patience time is negative or NaN.

```
public ServiceTimes getContactTimes()
```

Returns the contact times for this contact.

Returns the contact times.

```
public ServiceTimes getAfterContactTimes()
```

Returns the after-contact times for this contact.

Returns the after-contact times.

```
public double getDefaultServiceTime()
```

Returns the default service time for this contact object. This corresponds to the result of the sum of `getDefaultContactTime()`, and `getDefaultAfterContactTime()`.

Returns the service time of the contact.

```
public void setDefaultServiceTime (double serviceTime)
```

Sets the default service time of this contact to `serviceTime`. This method sets the contact time to `serviceTime`, and resets the after-contact time to 0.

Parameter

`serviceTime` the new service time of the contact.

Throws

`IllegalArgumentException` if the given service time is negative or NaN.

```
public double getWaitingTimeEstimate()
```

Returns the estimate waiting time that a call must before beginning its service in the call center.

Returns the waiting time estimate for a call when it arrive in the call center.

```
public void setWaitingTimeEstimate (double waitingTimeEstimate)
```

Sets the waiting time estimate value for a call arrive in the call center. to `waitingTimeEstimate`.

Parameter

`waitingTimeVQ` the new waiting time in virtual queue.

```
public double getDefaultContactTime()
```

Returns the default contact time with an agent. By default, this is set to `Double.POSITIVE_INFINITY`.

Returns the default contact time.

```
public void setDefaultContactTime (double contactTime)
```

Sets the default contact time to `contactTime`.

Parameter

`contactTime` the new contact time.

Throws

`IllegalArgumentException` if the contact time is negative or NaN.

```
public double getDefaultAfterContactTime()
```

Returns the default duration of after-contact work performed by an agent after this contact is served. By default, this is set to 0.

Returns the default after-contact time.

```
public void setDefaultAfterContactTime (double afterContactTime)
```

Sets the default after-contact time to `afterContactTime`.

Parameter

`afterContactTime` the new after-contact time.

Throws

`IllegalArgumentException` if the after-contact time is negative or NaN.

```
public double getDefaultContactTime (int i)
```

Returns the default contact time if this contact is served by an agent in group `i`. If this contact time was never set, this returns the result of `getDefaultContactTime()`.

Parameter

`i` the index of the agent group.

Returns the contact time.

```
public boolean isSetDefaultContactTime (int i)
```

Determines if a contact time was set specifically for agent group *i*, by using `setDefaultContactTime (int, double)`.

Parameter

i the tested agent group index.

Returns the result of the test.

```
public void setDefaultContactTime (int i, double t)
```

Sets the default contact time for this contact if served by an agent in group *i* to *t*. Note that setting *t* to `Double.NaN` unsets the contact time for the specified agent group.

Parameters

i the index of the agent group to set.

t the new contact time.

```
public void ensureCapacityForDefaultContactTime (int capacity)
```

Makes sure that the array containing default contact times specific to each agent group contains at least *capacity* elements. This method should be called before `setDefaultContactTime (int, double)` to avoid multiple array reallocation.

Parameter

capacity the new capacity.

```
public double getDefaultAfterContactTime (int i)
```

Returns the default after-contact time if this contact is served by an agent in group *i*. If this after-contact time was never set, this returns the result of `getDefaultAfterContactTime()`.

Parameter

i the index of the agent group.

Returns the after-contact time.

```
public boolean isSetDefaultAfterContactTime (int i)
```

Determines if an after-contact time was set specifically for agent group *i*, by using `setDefaultAfterContactTime (int, double)`.

Parameter

i the tested agent group index.

Returns the result of the test.

```
public void setDefaultAfterContactTime (int i, double t)
```

Sets the default after-contact time for this contact to `t`, if served by an agent in group `i`. Note that setting `t` to `Double.NaN` unsets the after-contact time for the specified agent group.

Parameters

`i` the index of the agent group to set.

`t` the new after-contact time.

```
public void ensureCapacityForDefaultAfterContactTime (int capacity)
```

Makes sure that the array containing default after-contact times specific to each agent group contains at least `capacity` elements. This method should be called before `setDefaultAfterContactTime (int, double)` to avoid multiple array reallocation.

Parameter

`capacity` the new capacity.

```
public double getDefaultServiceTime (int i)
```

Returns the default service time for this contact if served by an agent in group `i`. This returns the sum of `getDefaultContactTime (int)` and `getDefaultAfterContactTime (int)`.

Parameter

`i` the tested agent group.

Returns the default service time.

```
public List<ContactStepInfo> getSteps()
```

Returns the list containing the steps in the life cycle of this contact. This list should contain `ContactStepInfo` implementations only. If steps tracing was not enabled for this contact, this returns `null`.

Returns the list of contact steps, or `null` if steps tracing is disabled.

```
public void enableStepsTracing()
```

Enables steps tracing for this contact object. By default, steps tracing is disabled for better performance and memory usage. This method should be called as soon as the contact is constructed to avoid any loss of information.

```
public double getPriority()
```

Returns the priority for this contact. The priority is a number which indicates the level of emergency of the contact. A low value represents a high priority.

Returns the contact's priority.

```
public void setPriority (double newPriority)
```

Changes the contact's priority to `newPriority`. This method should not be called when a contact is in a priority queue.

Parameter

`newPriority` the new contact's priority.

```
public double getTotalQueueTime()
```

Returns the total time the contact has spent waiting in queues. This returns the cumulative waiting time, for all waiting queues visited by the contact.

Returns the contact's queue time.

```
public void addToTotalQueueTime (double delta)
```

Adds `delta` to the currently recorded total queue time returned by `getTotalQueueTime()`. This method can be used, e.g., to subtract time passed in a virtual queue from the queue time of a contact.

Parameter

`delta` the amount to add.

```
public WaitingQueue getLastWaitingQueue()
```

Returns the last waiting queue this contact entered in. If the contact was never queued, this returns `null`.

Returns the last waiting queue of the contact.

```
public double getTotalServiceTime()
```

Returns the total time this contact has spent being served by agents. This returns the cumulative contact time (not after-contact time) for all agents visited by the contact.

Returns the contact's service time.

```
public void addToTotalServiceTime (double delta)
```

Adds `delta` to the currently recorded total service time returned by `getTotalServiceTime()` for this contact.

Parameter

`delta` the amount to add.

```
public AgentGroup getLastAgentGroup()
```

Returns the last agent group who began serving this contact. If the contact was never served, this returns `null`.

Returns the last agent group serving the contact.

```
public int getId()
```

Returns the type identifier for this contact object.

Returns the contact's type identifier.

```
public void setId (int newId)
```

Changes the type identifier for this contact object to `newId`. The type identifier of a contact should not change when it is in a waiting queue or served by an agent.

Parameter

`newId` the contact's new type identifier.

Throws

`IllegalArgumentException` if the type identifier is smaller than 0.

```
public Map<Object, Object> getAttributes()
```

Returns the map containing the attributes for this contact. Attributes can be used to add user-defined information to contact objects at runtime, without creating a subclass. However, for maximal efficiency, it is recommended to create a subclass of `Contact` instead of using attributes.

Returns the map containing the attributes for this object.

```
public ContactSource getSource()
```

Returns the contact's primary source which has produced this contact object. If no source has created this contact, this returns `null`. If a contact results from a call back managed by a dialer, this returns the preceding arrival process which created the contact, not the dialer managing the call back.

Returns the source having created this contact.

```
public void setSource (ContactSource src)
```

Sets the source of this contact to `src`. Once a non-null source was given, it cannot be changed. If one tries to change the contact source, an `IllegalStateException` is thrown.

Parameter

`src` the new contact source.

Throws

`IllegalStateException` if one tries to change the contact source.

```
public TrunkGroup getTrunkGroup()
```

Returns the trunk group this contact will take a trunk from. By default, this returns `null`.

Returns the associated trunk group.

```
public void setTrunkGroup (TrunkGroup tg)
```

Sets the trunk group for this contact to **tg**. This method does not allocate a trunk in the group; this task is performed by the router.

Parameter

tg the new trunk group.

```
public Router getRouter()
```

Returns a reference to the router currently managing this contact, or **null** if the contact is not currently in a router.

Returns the router taking care of this contact.

```
public void setRouter (Router router)
```

This should only be called by the router. Associates the router **router** to this contact.

Parameter

router the new router.

```
public boolean hasExited()
```

Determines if the contact has exited the system. If a contact has exited the system, it will not be admitted into a router for further processing. This is used to prevent contacts from incorrectly entering in the router several times.

Returns the exited indicator.

```
public void setExited (boolean b)
```

Sets the exited indicator to **b**.

Parameter

b the exited indicator.

```
public void enqueued (DequeueEvent ev)
```

This method is called by a waiting queue object when a contact is put in queue, the dequeue event **ev** representing the queued contact.

Parameter

ev the dequeue event associated with the contact.

```
public void dequeued (DequeueEvent ev)
```

This method is called when a contact leaves a queue, the dequeue event **ev** representing the queued contact.

Parameter

ev the dequeue event associated with the contact.

```
public void blocked (int bType)
```

This method is called when the contact is blocked by its current router with blocking type **bType**. The `getRouter()` method can be used to access the reference to the router which blocked this contact while **bType** indicates the reason why the contact was blocked.

Parameter

bType the contact blocking type.

```
public void beginService (EndServiceEvent ev)
```

This method is called when the service of this contact by an agent begins, the end-service event **ev** representing the contact being served.

Parameter

ev the event occurring at the end of the service.

```
public void endContact (EndServiceEvent ev)
```

This method is called when the communication between this contact and an agent is terminated. The end-service event **ev** can be used to obtain information about the end of communication.

Parameter

ev the end-service event associated with the contact.

```
public void endService (EndServiceEvent ev)
```

This method is called when the service of this contact (communication and after-contact work) was terminated, **ev** containing information about the served contact.

Parameter

ev the end-service event associated with the contact.

```
public int getNumWaitingQueues()
```

Returns the number of waiting queues this contact is waiting in simultaneously, at the current simulation time.

Returns the number of waiting queues this contact is waiting in.

```
public int getNumAgentGroups()
```

Returns the number of agent groups serving this contact simultaneously at the current simulation time.

Returns the number of agent groups serving this contact simultaneously.

```
public int compareTo (Contact otherContact)
```

Compares this contact with **otherContact**. By default, the contacts are ordered in ascending order of priority. The lower the priority value, the more important the contact will be. If two compared contacts share the same priority, they are ordered using their arrival times.

Parameter

otherContact the other contact this contact is compared to.

Returns a value smaller than 0 if this contact is greater than the other object, 0 if it is equal, or a value greater than 0 if it is smaller.

```
public Contact clone()
```

Returns a copy of this contact object. In contrast with the original contact object, the returned copy is not in any waiting queue, router, or agent group. The map containing the attributes, if **getAttributes()** returns a non-**null** value, is cloned, but the elements in the map are not cloned. If contact steps tracing is enabled, the list of steps as well as the step objects are cloned.

Returns the copy of the contact.

ServiceTimes

Stores service times for a contact. By default, there are two types of service times: contact times and after-contact times. However, a model can define additional types of service times. For each of these types, one may generate a default service time v which applies for all agents, one service time v_i for each agent group i . This class can be used to store and retrieve such service times. For greater efficiency, it is recommended to call method `ensureCapacityForServiceTime` before using `setServiceTime` in order to avoid multiple array reallocations.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class ServiceTimes implements Cloneable
```

Constructor

```
public ServiceTimes (double serviceTime)
```

Constructs a new container for service times using the default service time `serviceTime`. It is used when no service time is given for a specific agent group.

Parameter

`serviceTime` the default service time v .

Throws

`IllegalArgumentException` if the given service time is negative or NaN.

Methods

```
public double getServiceTime()
```

Returns the default service time v for this object.

Returns the default service time.

```
public void setServiceTime (double serviceTime)
```

Sets the default service time v of this object to `serviceTime`.

Parameter

`serviceTime` the new default service time.

Throws

`IllegalArgumentException` if the given service time is negative or NaN.

```
public double getServiceTime (int i)
```

Returns the service time v_i for contacts served by an agent in group i . If this service time was never set, i.e., if `isSetServiceTime (int)` returns `false`, this returns the result of `getServiceTime()`.

Parameter

i the index of the agent group.

Returns the service time v_i , or v if v_i is not set.

```
public double[] getServiceTimes()
```

Returns the array of service times for all groups.

Returns the service times of all groups

```
public boolean isSetServiceTime (int i)
```

Determines if a service time was set specifically for agent group *i*, by using `setServiceTime (int, double)`.

Parameter

i the tested agent group index.

Returns the result of the test.

```
public void setServiceTime (int i, double t)
```

Sets the service time v_i for contacts served by an agent in group *i* to *t*. Note that setting *t* to `Double.NaN` unsets the service time for the specified agent group.

Parameters

i the index of the agent group to set.

t the new service time.

Throws

`IllegalArgumentException` if *t* is negative.

```
public void ensureCapacityForServiceTime (int capacity)
```

Makes sure that the length of the array containing the v_i 's is at least *capacity* for the number of groups. This method should be called before `setServiceTime (int, double)` to avoid multiple array reallocations.

Parameter

capacity the new capacity for the groups

```
public void set (ServiceTimes st)
```

Replaces the service times v and v_i 's stored in this object with the values obtained from *st*.

Parameter

st the input service times.

```
public void add (ServiceTimes st)
```

Adds the service times stored in *st* to the corresponding service times in this object. Let v and v_i for $i = 0, \dots$, be the service times in this object, and w and w_i , the service times in *st*. This method replaces v with $v + w$; and v_i with $v_i + w_i$ for any i such that v_i or w_i exists. When v_i or w_i does not exist, v or w is used.

Parameter

`st` the service times to add to this object.

`public void mult (double mult)`

Multiplies each service time v and v_i stored in this object by the given constant `mult`.

Parameter

`mult` the multiplier for service times.

Throws

`IllegalArgumentException` if `mult` is negative.

`public ServiceTimes clone()`

Clones this object, and its internal arrays of service times.

ContactFactory

Allows contact sources to create contact objects of user-defined classes. When the **Contact** class is extended to add user-defined attributes, a contact factory must also be created to allow contact sources to instantiate objects derived from the **Contact** subclass. To construct a new contact factory, the user simply implements this interface to provide a **newInstance()** method the contact sources call to get contacts.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public interface ContactFactory
```

Method

```
public Contact newInstance()
```

Constructs and returns a new **Contact** object. If a contact cannot be instantiated, a **ContactInstantiationException** is thrown.

Returns the new contact object.

Throws

ContactInstantiationException if a contact cannot be instantiated.

SimpleContactFactory

This implements the `ContactFactory` interface to instantiate `Contact` objects with fixed parameters.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class SimpleContactFactory implements ContactFactory
```

Constructors

```
public SimpleContactFactory()
```

Constructs a new contact factory which will create contact objects with priority 1 and type ID 0.

```
public SimpleContactFactory (Simulator sim)
```

Equivalent to `SimpleContactFactory()`, using the given simulator `sim`.

```
public SimpleContactFactory (int typeId)
```

Constructs a new contact factory which will create contact objects with priority 1 and type ID `typeId`.

Parameter

`typeId` the type ID of the contacts.

```
public SimpleContactFactory (Simulator sim, int typeId)
```

Equivalent to `SimpleContactFactory (int)`, using the given simulator `sim`.

```
public SimpleContactFactory (double priority, int typeId, boolean tracing)
```

Constructs a new contact factory which will create contact objects with priority `priority` and type ID `typeId`. If `tracing` is `true`, contact objects with steps tracing enabled will be created.

Parameters

`priority` the priority of the contact.

`typeId` the type ID of the contacts.

`tracing` the contact steps tracing indicator.

```
public SimpleContactFactory (Simulator sim, double priority, int typeId,  
                             boolean tracing)
```

Equivalent to `SimpleContactFactory (double, int, boolean)`, using the given simulator `sim`.

Methods

```
public Simulator simulator()
```

Returns the simulator associated with this contact factory. This simulator is associated with every contact instantiated by the factory.

Returns the associated simulator.

```
public void setSimulator (Simulator sim)
```

Sets the simulator associated with this contact factory to `sim`.

Parameter

`sim` the new associated simulator.

```
public double getPriority()
```

Returns the priority of the created and reused contact objects.

Returns the priority of the generated contact.

```
public int getTypeId()
```

Returns the type ID of the created and reused contact objects.

Returns the type ID of the generated contact.

```
public boolean getTracing()
```

Returns `true` if the created contacts will support steps tracing.

Returns the contact steps tracing indicator.

SingleTypeContactFactory

Represents a contact factory used to create contacts of a single type. This factory also associates default patience, contact, and after-contact times to the constructed contacts. All random variates are generated at the time the contact is created.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class SingleTypeContactFactory implements ContactFactory
```

Constructors

```
public SingleTypeContactFactory (int type, ValueGenerator probBalkGen,
                                RandomStream streamBalk, RandomVariateGen
                                pgen, RandomVariateGen cgen,
                                RandomVariateGen[] cgenGroups,
                                RandomVariateGen acgen, RandomVariateGen[]
                                acgenGroups)
```

Constructs a new contact factory constructing contacts of type `type`. The `probBalkGen` value generator is used to generate probabilities of balking while `streamBalk` is used to determine if the contact balks if not served immediately. The generators `pgen`, `cgen`, and `acgen` are used to generate patience times for contacts that do not balk, contact times, and after-contact times. `cgenGroups` and `acgenGroups` can be used to generate contact and after contact times used if the contact is served by a specific agent group.

If `probBalkGen` or `streamBalk` are `null`, the probability of balking will always be 0. If `pgen` is `null`, the patience time will always be infinite. The default contact time when the given generator is `null` is infinite while the default after-contact time is 0.

The constructed call factory assigns the default simulator returned by `Simulator.getDefaultSimulator()` to each new contact.

Parameters

`type` the contact type identifier of all new contacts.

`probBalkGen` the generator for balking probabilities.

`streamBalk` the random stream for balking.

`pgen` the patience time generator.

`cgen` the default contact time generator.

`cgenGroups` the agent-group specific contact time generators.

`acgen` the default after-contact time generator.

`acgenGroups` the agent-group specific after-contact time generators.

```
public SingleTypeContactFactory (Simulator sim, int type, ValueGenerator
                                probBalkGen, RandomStream streamBalk,
                                RandomVariateGen pgen, RandomVariateGen
                                cgen, RandomVariateGen[] cgenGroups,
                                RandomVariateGen acgen, RandomVariateGen[]
                                acgenGroups)
```

Equivalent to `SingleTypeContactFactory (int, ValueGenerator, RandomStream, RandomVariateGen, RandomVariateGen, RandomVariateGen[], RandomVariateGen, RandomVariateGen[])`, using the given simulator `sim`.

Methods

```
public Simulator simulator()
```

Returns the simulator associated with this contact factory. This simulator is associated with every contact instantiated by the factory.

Returns the associated simulator.

```
public void setSimulator (Simulator sim)
```

Sets the simulator associated with this contact factory to `sim`.

Parameter

`sim` the new associated simulator.

```
public Contact newInstance()
```

Creates a new instance of class `Contact`, and initializes it by calling the `setRandomVariables (Contact)` method.

```
public void setRandomVariables (Contact contact)
```

Generates the random variates related to a contact, and assigns the generated value to the given `contact` object.

Parameter

`contact` the contact object to set up.

```
public RandomStream getStreamBalk()
```

Returns the random stream used for balking.

Returns the random stream used for balking.

```
public void setStreamBalk (RandomStream streamBalk)
```

Sets the random stream used for balking to `streamBalk`.

Parameter

streamBalk the new random stream for balking.

```
public ValueGenerator getProbBalkGenerator()
```

Returns a reference to the value generator used for generating probabilities of balking.

```
public void setProbBalkGenerator (ValueGenerator probBalkGen)
```

Sets the value generator for probability of balking to **probBalkGen**.

```
public RandomVariateGen getPatienceTimeGen()
```

Returns the random-variate generator for patience times.

Returns the random variate generator for patience times.

```
public void setPatienceTimeGen (RandomVariateGen pgen)
```

Sets the random variate generator for patience times to **pgen**.

Parameter

pgen the new random variate generator for patience times.

```
public RandomVariateGen getContactTimeGen()
```

Returns the random-variate generator for default contact times. This generates the contact times used when no contact time specific to the agent group performing the service is available.

Returns the random variate generator for contact times.

```
public void setContactTimeGen (RandomVariateGen cgen)
```

Sets the random variate generator for default contact times to **cgen**.

Parameter

cgen the new random variate generator for contact times.

```
public RandomVariateGen getAfterContactTimeGen()
```

Returns the random-variate generator for default after-contact times. This generates the after-contact times used when no after-contact time specific to the agent group performing the service is available.

Returns the random variate generator for default after-contact times.

```
public void setAfterContactTimeGen (RandomVariateGen acgen)
```

Sets the random variate generator for default after-contact times to **acgen**.

Parameter

`acgen` the new random variate generator for default after-contact times.

```
public RandomVariateGen[] getContactTimeGenGroups()
```

Returns the random variate generators for contact times when served by agents in specific groups.

Returns the contact time generators.

```
public RandomVariateGen getContactTimeGen (int i)
```

Returns the random variate generator for contacts served by agents in group `i`.

Parameter

`i` the agent group index.

Returns the contact time generator.

```
public void setContactTimeGenGroups (RandomVariateGen[] cgenGroups)
```

Sets the contact-time generators for contacts served by specific agent groups to `cgenGroups`.

Parameter

`cgenGroups` the new contact-time generators.

```
public RandomVariateGen[] getAfterContactTimeGenGroups()
```

Returns the random variate generators for after-contact times when served by agents in specific groups.

Returns the after-contact time generators.

```
public RandomVariateGen getAfterContactTimeGen (int i)
```

Returns the random variate generator for contacts served by agents in group `i`.

Parameter

`i` the agent group index.

Returns the after-contact time generator.

```
public void setAfterContactTimeGenGroups (RandomVariateGen[] acgenGroups)
```

Sets the contact-time generators for contacts served by specific agent groups to `cgenGroups`.

Parameter

`acgenGroups` the new contact-time generators.

```
public double getMeanContactTime (int i)
```

Returns the mean contact time for a new contact served by an agent in group `i`.

Parameter

`i` the agent group identifier.

Returns the mean contact time.

```
public double getMeanAfterContactTime (int i)
```

Returns the mean after-contact time for a new contact served by an agent in group `i`.

Parameter

`i` the agent group identifier.

Returns the mean contact time.

```
public int getId()
```

Returns the type identifier for contacts returned by this factory.

Returns the type identifier of constructed contacts.

```
public void setId (int type)
```

Sets the type identifier of constructed contacts to `type`.

Parameter

`type` the type identifier of constructed contacts.

RandomTypeContactFactory

Represents a contact factory that can create contacts of random types. Any instance of this class encapsulates an array of contact factories, and a probability of selection for each factory. Each time a new contact is needed, an internal factory is selected randomly based on the selection probabilities, and used to instantiate the contact.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class RandomTypeContactFactory implements ContactFactory
```

Constructor

```
public RandomTypeContactFactory (ContactFactory[] factories, double[] prob,  
                                RandomStream stream)
```

Constructs a random-type contact factory selecting contact factories from the array **factories**, with probabilities given by **prob**, and using the random stream **stream**. Contact factory **factories[k]** is selected with probability **prob[k]**, for **k=0,...,factories.length-1**. If the values in **prob** do not sum to 1, they are normalized by dividing by their sum.

Parameters

factories the array of contact factories.

prob the array of probabilities of selection.

stream the random stream.

Throws

NullPointerException if any of the above argument is **null**, or if at least one given contact factory is **null**.

IllegalArgumentException if **factories** and **prob** do not share the same length, or if **prob** contains at least one negative value.

Methods

```
public ContactFactory[] getContactFactories()
```

Returns an array giving each internal contact factory that can be selected.

Returns the array of contact factories.

```
public double[] getProbabilities()
```

Returns an array giving the probability of selection for each internal contact factory.

Returns the array containing probabilities of selection.

```
public RandomStream getStream()
```

Returns the random stream used for performing the selection.

Returns the random stream for selection.

```
public void setStream (RandomStream stream)
```

Sets the random stream for performing further selections to **stream**.

Parameter

stream the new random stream for selection.

Throws

`NullPointerException` if **stream** is null.

ContactInstantiationException

This exception is thrown when a contact factory cannot instantiate a contact on a call to its `ContactFactory.newInstance()` method.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class ContactInstantiationException extends RuntimeException
```

Constructors

```
public ContactInstantiationException (ContactFactory factory)
```

Constructs a new contact instantiation exception with no message, no cause, and thrown by the given `factory`.

Parameter

`factory` the contact factory having thrown the exception.

```
public ContactInstantiationException (ContactFactory factory, String  
                                     message)
```

Constructs a new contact instantiation exception with the given `message`, no cause, and concerning `factory`.

Parameters

`factory` the contact factory concerned by the exception.

`message` the error message describing the exception.

```
public ContactInstantiationException (ContactFactory factory, Throwable  
                                     cause)
```

Constructs a new contact instantiation exception with no message, the given `cause`, and concerning `factory`.

Parameters

`factory` the contact factory concerned by the exception.

`cause` the cause of the exception.

```
public ContactInstantiationException (ContactFactory factory, String  
                                     message, Throwable cause)
```

Constructs a new contact instantiation exception with the given `message`, the supplied `cause`, and concerning `factory`.

Parameters

factory the contact factory concerned by the exception.

message the error message describing the exception.

cause the cause of the exception.

Methods

```
public ContactFactory getContactFactory()
```

Returns the contact factory concerned by this exception.

Returns the contact factory concerned by this exception.

```
public String toString()
```

Returns a short description of the exception. If `getContactFactory()` returns `null`, this calls `super.toString`. Otherwise, the result is the concatenation of the strings:

- The name of the actual class of the exception
- `": For contact factory "`
- the result of `getContactFactory().toString()`
- If `Throwable.getMessage()` is non-null
 - `", "`
 - The result of `Throwable.getMessage()`

Returns a string representation of the exception.

NewContactListener

Defines a new-contact listener that receives incoming contacts for further processing. This interface is mainly used to link the contact arrival processes to routers or dialer lists. It can also be used for counting the number of arrivals, for statistical collecting.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public interface NewContactListener
```

Method

```
public void newContact (Contact contact)
```

Notifies the listener about a new contact `contact`. The given contact object can be assumed non-null, and may be stored or processed in any needed ways.

Parameter

`contact` the new contact.

ContactSumMatrix

This sum matrix can be used to compute contact-related observations. It defines one measure type for each contact type as well as one aggregate measure. When the supported number of contact types is 1, it computes the aggregate sum only. When the supported number of contact types is greater than 1, it computes a sum specific for each type as well as the aggregate sum. If one does not require the sum row, one can use a `SumMatrix` instead.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class ContactSumMatrix extends SumMatrix
```

Constructors

```
public ContactSumMatrix (int numTypes)
```

Constructs a new contact sum matrix for `numTypes` contact types and one period.

Parameter

`numTypes` the number of contact types.

Throws

`IllegalArgumentException` if the number of contact types is negative or 0.

```
public ContactSumMatrix (PeriodChangeEvent pce, int numTypes)
```

Constructs a new contact sum matrix with period change event `pce` and for `numTypes` contact types. The number of periods is determined by using `PeriodChangeEvent.getNumPeriods()`.

Parameters

`pce` the period change event.

`numTypes` the number of contact types.

Throws

`IllegalArgumentException` if the number of contact types is negative or 0.

`NullPointerException` if `pce` is null.

```
public ContactSumMatrix (int numTypes, int numPeriods)
```

Constructs a new contact sum matrix for `numTypes` contact types and `numPeriods` periods.

Parameters

`numTypes` the number of contact types.

`numPeriods` the number of periods.

Throws

`IllegalArgumentException` if the number of contact types or periods is negative or 0.

Methods

```
public void add (Contact contact, double x)
```

Equivalent to `add (contact.getTypeId(), period, x)` where `period` is the period at which the contact arrived. If no period change event was associated with this object, the period is always 0.

Parameters

`contact` the contact to which the observation is related.

`x` the value being added.

Throws

`NullPointerException` if `contact` is null.

```
public void add (Contact contact, int period, double x)
```

Equivalent to `add (contact.getTypeId(), period, x)`.

Parameters

`contact` the contact to which the observation is related.

`period` the period the observation is added to.

`x` the value being added.

Throws

`NullPointerException` if `contact` is null.

```
public void add (int type, int period, double x)
```

Adds a new observation `x` for contact type `type` in the period `period`. If the object supports only one contact type, this will add one observation in the measure 0 of the matrix, independently of the contact type identifier. Otherwise, an observation is added in the measure corresponding the contact type as well as in the last measure for the aggregate sum. Even if the contact type identifier cannot be mapped to a valid measure index, the observation is added to the last row.

Parameters

`type` the contact type of the new observation.

`period` the period of the new observation.

`x` the value being added.

Throws

`ArrayIndexOutOfBoundsException` if `type` or `period` are negative or greater than or equal to the number of supported contact types or periods.

ContactStepInfo

Represents an information object about a single step (end of service, exit of waiting queue, etc.) in the life cycle of a contact in the contact center. Implementations of this interface are used when the steps of contacts are traced.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public interface ContactStepInfo
```

Methods

```
public Contact getContact()
```

Returns the contact concerned by this step.

Returns the concerned contact.

```
public double getStartingTime()
```

Returns the simulation time at which this step started.

Returns the start time of the step.

```
public double getEndingTime()
```

Returns the simulation time at which this step ended.

Returns the end time of the step.

```
public ContactStepInfo clone (Contact clonedContact)
```

Makes a copy of this data object that will be associated with the cloned contact `clonedContact`. This method is intended to be used in `Contact.clone()`.

Parameter

`clonedContact` the contact being cloned.

Returns the clone of this data object.

TrunkGroup

Represents a group of trunks, i.e., phone lines or more generally communication channels, in a contact center. After a contact is constructed, it can be assigned a trunk group using `Contact.setTrunkGroup (TrunkGroup)`. When the contact enters the router, a line is allocated. The contact is blocked if a line is not available.

```
package umontreal.iro.lecuyer.contactcenters.contact;  
  
public class TrunkGroup implements Initializable, Named
```

Constructor

```
public TrunkGroup (int capacity)
```

Constructs a new trunk group with capacity `capacity`. The capacity corresponds to the maximum number of allocated lines at any simulation time.

Parameter

`capacity` the total number of lines in the trunk group.

Throws

`IllegalArgumentException` if the capacity is negative.

Methods

```
public int getCapacity()
```

Returns the current capacity of this trunk group.

Returns the current capacity.

```
public void setCapacity (int capacity)
```

Changes the capacity to `capacity`. If the given capacity is negative or smaller than the current number of allocated lines, an `IllegalArgumentException` is thrown.

Parameter

`capacity` the new capacity.

Throws

`IllegalArgumentException` if capacity is too small or negative.

```
public int lines()
```

Returns the current number of allocated lines.

Returns the current number of lines.

```
public void init()
```

Resets this trunk group, releasing all allocated lines. If statistical collecting is enabled, this also calls `initStat()`.

```
public void initStat()
```

Initializes the two statistical collectors for the number of lines and the capacity. If statistical collecting is disabled, this throws an `IllegalStateException`.

Throws

`IllegalStateException` if statistical collecting is disabled.

```
public boolean take (Contact contact)
```

Indicates that the contact `contact` enters the system and takes one line from this trunk group. If all lines are busy, this returns `false`. Otherwise, this returns `true`.

Parameter

`contact` the contact allocating the line.

Returns the success indicator.

```
public void release (Contact contact)
```

Releases the trunk line allocated by the contact `contact`.

Parameter

`contact` the contact releasing the line.

```
public boolean isStatCollecting()
```

Determines if this trunk group is collecting statistics about the number of allocated lines and its capacity. By default, statistical collecting is turned OFF.

Returns the statistical collecting indicator.

```
public void setStatCollecting (boolean b)
```

Sets the statistical collecting to `b`. If `b` is `true`, the collecting is turned ON. Otherwise, it is turned OFF.

Parameter

`b` the statistical collecting indicator.

```
public void setStatCollecting (Simulator sim)
```

Enables statistical collecting, but associates the given simulator to the internal accumulates.

Parameter

`sim` the simulator associated to the internal accumulates.

```
public void setStatCollectiong (Simulator sim)
```

Enables statistical collecting, and attach the simulator `sim` to the internal accumulates. The given simulator is used to determine the simulation time when the values of the probes are updated.

Parameter

`sim` the given simulator.

```
public Accumulate getStatCapacity()
```

Returns the statistical collector for the capacity of this trunk group through simulation time. The returned value is non-null only if `setStatCollecting (boolean)` was called with `true`.

Returns the statistical collector for the capacity of this trunk group.

```
public Accumulate getStatLines()
```

Returns the statistical collector for the number of allocated lines through simulation time. The returned value is non-null only if `setStatCollecting (boolean)` was called with `true`.

Returns the statistical collector for the number of allocated lines.

ContactSource

Represents a contact source which produces contacts during a simulation. Before any simulation replication, any contact source needs to be initialized. Since initialization disables the source, the source must be enabled to produce contacts. When a contact is produced, the contact source should instantiate a `Contact` object using a user-specified `ContactFactory` implementation, or pick an instance from an internal list. It should then notify the new contact to any registered `NewContactListener` implementation.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public interface ContactSource extends ToggleElement, Initializable, Named
```

Methods

```
public void init()
```

Initializes the contact source for a new replication of a simulation. This method should disable the contact source if it is enabled, and cancel any scheduled event. One can assume this method will be called before any simulation replication starts.

```
public void addNewContactListener (NewContactListener listener)
```

Adds the listener `listener` to be notified when a new contact is produced. If the listener was already registered, nothing happens, because the listener cannot be notified more than once.

Parameter

`listener` the new-contact listener being added.

Throws

`NullPointerException` if `listener` is `null`.

```
public void removeNewContactListener (NewContactListener listener)
```

Removes the new-contact listener `listener` from the list associated with this contact source. If the listener was not previously registered with this contact source, nothing happens.

Parameter

`listener` the new-contact listener being removed.

```
public void clearNewContactListeners()
```

Clears the list of new-contact listeners associated with this contact source.

```
public List<NewContactListener> getNewContactListeners()
```

Returns an unmodifiable list containing all the new-contact listeners registered with this contact source.

Returns the list of all registered new-contact listeners.

```
public Simulator simulator()
```

Returns a reference to the simulator associated with this contact source. The simulator is used to schedule any event required by the contact source to produce contacts.

Any implementation of this interface should provide a constructor accepting the simulator as an argument. Constructors not receiving a simulator should use the default simulator returned by `Simulator.getDefaultSimulator()`.

Returns the associated simulator.

```
public void setSimulator (Simulator sim)
```

Sets the simulator attached to this contact source to `sim`. This method should not be called while the contact source is started.

Parameter

`sim` the new simulator.

Throws

`NullPointerException` if `sim` is null.

ContactArrivalProcess

Represents a contact arrival process modeling the arrival of inbound contacts. Such a process schedules an event for each new contact, and broadcasts the arrival to any registered new-contact listeners. More specifically, a single simulation event manages arrivals as follows: upon an arrival, a new contact is instantiated using the associated contact factory, the contact is broadcast to any registered listener, and the next arrival is scheduled. The interarrival times are computed using the `nextTime()` method which needs to be implemented in a concrete subclass. This abstract class also takes care of new-contact listeners registration and notification. Subclasses only needs to define `nextTime()` and optionally `init()` which initializes the arrival process at the beginning of the simulation. It is also possible to access the scheduled new-contact event to reschedule or cancel it as needed. Implementing `getArrivalRate (int)`, and `getExpectedArrivalRate (int)` is recommended to allow programs to get the arrival rate and expected arrival rate.

Each arrival process has an associated simulator which is an instance of the `Simulator` class. This simulator is used to schedule the event managing the arrival process. It is also assumed that the user-defined contact factory attaches this simulator to each new contact. Failing to meet this condition might lead to unexpected behavior, and will trigger a failed assertion if assertion checking is turned on during execution.

The arrival process can be inflated or deflated by a *busyness factor* denoted B , a random variable with mean 1, and usually generated once for a day. Any arrival process can be defined as $\{N(t), t \geq 0\}$, where $N(t)$ is the number of arrivals during the time interval $[0, t)$. The process affected by the busyness, $\{\tilde{N}(t), t \geq 0\}$, is given by taking $\tilde{N}(t) = \text{round}(BN(t))$, where $\text{round}(\cdot)$ rounds its argument to the nearest integer. The exact way to take account of the busyness factor depends on the specific arrival process. For example, for Poisson processes, the busyness is used to inflate or deflate the λ arrival rate.

The busyness factor must be set externally, because the value of B for this arrival process is often correlated with B for other arrival processes. The recommended way to set B is using `init (double)`. The current value of B might be obtained using `getBusynessFactor()`. By default, it is assumed that $\mathbb{E}[B] = 1$. If this is not true for a particular model, one should call `setExpectedBusynessFactor (double)` to set the expectation of the factor.

Note: the `NewContactListener` implementations are notified in the order of the list returned by `getNewContactListeners()`, and a new-contact listener modifying the list of listeners by using `addNewContactListener (NewContactListener)` or `removeNewContactListener (NewContactListener)` could result in unpredictable behavior.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public abstract class ContactArrivalProcess implements ContactSource
```


Field

`protected final Event contactEvent`

Event representing the arrival of a new contact. Subclasses can cancel or reschedule this event to adjust it when a parameter change occurs.

Constructors

`public ContactArrivalProcess (ContactFactory factory)`

Constructs a new contact arrival process creating contacts using the given `factory`.

Parameter

`factory` the factory creating contacts for this arrival process.

Throws

`NullPointerException` if `factory` is `null`.

`public ContactArrivalProcess (Simulator sim, ContactFactory factory)`

Equivalent to `ContactArrivalProcess (ContactFactory)`, with a user-defined simulator `sim`.

Parameters

`sim` the simulator attached to this arrival process.

`factory` the factory creating contacts for this arrival process.

Throws

`NullPointerException` if `sim` or `factory` are `null`.

Methods

`public abstract double nextTime()`

Computes and returns the time before the next contact arrival is simulated by this object. If this method returns `Double.POSITIVE_INFINITY`, no more arrival events will be scheduled until the arrival process is reinitialized.

Returns the time before the next arrival.

`public double getBusynessFactor()`

Returns the currently used busyness factor B , which must be greater than or equal to 0, and defaults to 1.

Returns the current busyness factor.

```
public void setBusynessFactor (double b)
```

Sets the busyness factor to **b**. This method should be called before `init()` is called, or one should use `init (double)`.

Parameter

b the new busyness factor.

Throws

`IllegalArgumentException` if **b** is negative.

```
public double getExpectedBusynessFactor()
```

Returns the expected value of the busyness factor for this arrival process.

Returns the expected value of the busyness factor.

```
public void setExpectedBusynessFactor (double bMean)
```

Sets the expected busyness factor for this arrival process to **bMean**.

Parameter

bMean the new value of the expectation.

Throws

`IllegalArgumentException` if **bMean** is negative.

```
public void init (double b)
```

Initializes this process with a specific busyness factor $B = b$. By default, this method simply calls `setBusynessFactor (double)` followed by `init()`.

Parameter

b the value of the busyness factor.

Throws

`IllegalArgumentException` if $b \leq 0$.

```
public void init()
```

Initializes the new arrival process. If this method is overridden by a subclass, it is important to call `super.init()` in order to ensure that everything is initialized correctly.

```
public ContactFactory getContactFactory()
```

Returns a reference to the associated contact factory. This factory is used to instantiate contact objects.

Returns the associated contact factory.

```
public void setContactFactory (ContactFactory factory)
```

Sets the contact factory to **factory**. This new contact factory will be used to instantiate future contact objects.

Parameter

factory the new contact factory.

Throws

NullPointerException if the given contact factory is **null**.

```
public void startStationary()
```

Setup the arrival process to be stationary, and starts it using the **start()** method. When an arrival process is started using this method, its parameters do not evolve with time. This can be useful, e.g., to simulate a single period as if it was infinite in the model. If the arrival process does not support stationary mode, this method throws an unsupported-operation exception. The default behavior of this method is to throw this exception.

```
public void start (double delay)
```

Starts this arrival process and schedules the first arrival to happen after **delay** simulation time units, independently of how **nextTime()** is implemented. If **delay** is set to **Double.POSITIVE_INFINITY**, no arrival is scheduled. Any subsequent inter-arrival times will be generated with **nextTime()** as usual.

Parameter

delay the first inter-arrival time.

```
public double getNextArrivalTime()
```

Returns the simulation time of the next arrival currently scheduled by this arrival process. If the arrival process is stopped or no arrival is scheduled, this returns a negative number.

Returns the arrival time of the next contact.

```
public double getArrivalRate (int p)
```

Determines the arrival rate in period **p** for this arrival process. The arrival rate corresponds to the expected number of arrivals per simulation time unit during the specified period; one must multiply the rate by the period duration to get the expected number of arrivals during the period.

If arrival rate is random, this returns the arrival rate for the current replication. One should use **getExpectedArrivalRate (int)** or **getExpectedArrivalRateB (int)** to get the expected arrival rate.

If the arrival rate is not available, throws an **UnsupportedOperationException**.

Parameter

p the queried period index.

Returns the arrival rate in that period.

```
public void getArrivalRates (double[] rates)
```

Fills the given array `rates` with the arrival rate for each period. After this method returns, element `rates[p]` corresponds to the value returned by `getArrivalRate (p)`.

Parameter

`rates` the array filled with rates.

```
public double getExpectedArrivalRate (int p)
```

Determines the expected arrival rate in period `p` for this arrival process assuming that the expected value of the busyness factor is 1. The arrival rate corresponds to the expected number of arrivals per simulation time unit during the specified period; one must multiply the rate by the period duration to get the expected number of arrivals during the period. If arrival rates are deterministic, this returns the same value as `getArrivalRate (int)`.

If $\mathbb{E}[B] \neq 1$, one should use `getExpectedArrivalRateB (int)` which takes the expectation of the busyness factor into account.

If the expected arrival rate is not available, throws an `UnsupportedOperationException`. This is the default behavior of this method if not overridden by a subclass.

Parameter

`p` the queried period index.

Returns the expected arrival rate in that period.

```
public void getExpectedArrivalRates (double[] rates)
```

Fills the given array `rates` with the expected arrival rate for each period. After this method returns, element `rates[p]` corresponds to the value returned by `getExpectedArrivalRate (p)`.

Parameter

`rates` the array filled with rates.

```
public double getExpectedArrivalRateB (int p)
```

Returns the expected arrival rate considering the current expected busyness factor. This corresponds to the product of the value returned by `getExpectedArrivalRate (int)`, and the value returned by `getExpectedBusynessFactor()`.

Parameter

`p` the tested period.

Returns the tested arrival rate.

```
public void getExpectedArrivalRatesB (double[] rates)
```

Fills the given array `rates` with the expected arrival rate for each period. After this method returns, element `rates[p]` corresponds to the value returned by `getExpectedArrivalRateB (p)`.

Parameter

`rates` the array filled with rates.

```
public double getArrivalRate (double st, double et)
```

Determines the mean arrival rate in time interval $[s, e]$. The arrival rate corresponds to the expected number of arrivals per simulation time unit during the specified interval; one must multiply the rate by the interval length to get the expected number of arrivals during the interval. If $\lambda(t)$ is the arrival rate at time t , this method returns the result of

$$\int_s^e \lambda(t) dt / (e - s).$$

If arrival rate is random, this returns the arrival rate for the current replication. One should use `getExpectedArrivalRate (double, double)` or `getExpectedArrivalRateB (double, double)` to get the expected arrival rate.

This method returns 0 if $e \leq s$.

If the arrival rate is not available, throws an `UnsupportedOperationException`. This is the default behavior of this method if not overridden by a subclass.

Parameters

`st` the starting time s .

`et` the ending time e .

Returns the arrival rate in the given time interval.

```
public double getExpectedArrivalRate (double st, double et)
```

Determines the expected mean arrival rate in time interval $[s, e]$ for this arrival process assuming that the expected value of the busyness factor is 1. The arrival rate corresponds to the expected number of arrivals per simulation time unit during the specified interval; one must multiply the rate by the interval length to get the expected number of arrivals during the interval. If arrival rates are deterministic, this returns the same value as `getArrivalRate (double, double)`. If $\lambda(t)$ is the arrival rate at time t , this method returns

$$\int_s^e \mathbb{E}[\lambda(t)] dt / (e - s).$$

If $\mathbb{E}[B] \neq 1$, one should use `getExpectedArrivalRateB (double, double)` which takes the expectation of the busyness factor into account.

This method returns 0 if $e \leq s$.

If the expected arrival rate is not available, throws an `UnsupportedOperationException`. This is the default behavior of this method if not overridden by a subclass.

Parameters

`st` the starting time s .

`et` the ending time e .

Returns the expected arrival rate in the given time interval.

```
public double getExpectedArrivalRateB (double st, double et)
```

Returns the expected mean arrival rate considering the current expected busyness factor. This corresponds to the product of the value returned by `getExpectedArrivalRate(double, double)`, and the value returned by `getExpectedBusynessFactor()`.

Parameters

`st` the starting time s .

`et` the ending time e .

Returns the expected arrival rate in the given time interval.

```
public void notifyNewContact (Contact contact)
```

Notifies the contact `contact` to every registered listener.

Parameter

`contact` the contact to be notified.

StationaryContactArrivalProcess

Defines a contact arrival process with inter-arrival times following a stationary distribution. When an inter-arrival time is required, a random variate generator is used to get a random value.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class StationaryContactArrivalProcess extends ContactArrivalProcess
```

Constructors

```
public StationaryContactArrivalProcess (ContactFactory factory,  
                                         RandomVariateGen timesGen)
```

Constructs a new contact arrival process creating contacts using the given `factory` and using `timesGen` to generate the inter-arrival times.

Parameters

`factory` the factory creating contacts for this arrival process.

`timesGen` the random variate generator used to generate times between arrivals.

```
public StationaryContactArrivalProcess (Simulator sim, ContactFactory  
                                         factory, RandomVariateGen timesGen)
```

Equivalent to `StationaryContactArrivalProcess (ContactFactory, RandomVariateGen)`, using the given simulator `sim`.

Methods

```
public RandomVariateGen getTimesGen()
```

Returns the random variate generator used to generate the times between each arrival.

Returns the random variate generator associated with this object.

```
public void setTimesGen (RandomVariateGen timesGen)
```

Sets the random variate generator for inter-arrival times to `timesGen`.

Parameter

`timesGen` the new random variate generator.

Throws

`NullPointerException` if `timesGen` is null.

PoissonArrivalProcess

Represents a Poisson-based contact arrival process. This base class implements a Poisson arrival process with (piecewise-)constant arrival rates: when an inter-arrival time is required, it is generated from the exponential distribution with rate $B\lambda$. By default, the arrival rate is constant, but it may be changed at any time during the simulation. When the arrival rate changes, the currently scheduled arrival is adjusted automatically to reflect the change. This class can be used as a basis each time the rate function $\lambda(t)$ of a Poisson process is piecewise-constant over the simulation time t .

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class PoissonArrivalProcess extends ContactArrivalProcess
```

Constructors

```
public PoissonArrivalProcess (ContactFactory factory, double lambda,
                             RandomStream stream)
```

Constructs a new Poisson arrival process instantiating new contacts using **factory**. The parameter λ is initialized with **lambda** and the random number stream **stream** is used to generate the needed uniforms.

Parameters

factory the factory instantiating contacts.

lambda the initial value of $\lambda(t)$.

stream random number stream.

Throws

`IllegalArgumentException` if **lambda** < 0 .

`NullPointerException` if **factory** or **stream** are null.

```
public PoissonArrivalProcess (Simulator sim, ContactFactory factory,
                             double lambda, RandomStream stream)
```

Equivalent to `PoissonArrivalProcess (ContactFactory, double, RandomStream)`, with the given simulator **sim**.

Methods

```
public double getLambda()
```

Returns the current value of the arrival rate λ .

Returns the current value of λ .

```
public void setLambda (double newLambda)
```

Changes the value of λ to `newLambda`. This adjusts the time of the next arrival if necessary. If `newLambda` is set to 0, the currently scheduled arrival, if any, is cancelled and the Poisson process is stopped. The Poisson process can be restarted by setting a new non-zero λ value.

Parameter

`newLambda` the new value of λ .

Throws

`IllegalArgumentException` if `newLambda < 0`.

```
public RandomStream getStream()
```

Returns the random number stream used to generate the uniforms for inter-arrival times.

Returns the random number stream for the uniforms.

```
public void setStream (RandomStream stream)
```

Sets the random number stream used to generate the uniforms for the inter-arrival times to `stream`.

Parameter

`stream` the new random number stream.

Throws

`NullPointerException` if `stream` is null.

```
public boolean isCaching()
```

Determines if the generated inter-arrival times are cached for more efficiency. When caching is enabled, the arrival process records every standardized inter-arrival time generated. These random times follow the exponential distribution with $\lambda = 1$, and are divided by the arrival rate in use. Therefore, the cache can be used even if the arrival rate changes. The `initCache()` method must be called to start reusing cached values. This avoids some computations and increases the performance, at the expense of memory. This is useful when comparing several contact centers with common random numbers. By default, this caching is disabled for more efficient memory usage.

Returns the caching indicator for this arrival process.

```
public void setCaching (boolean caching)
```

Sets the caching indicator to `caching` for this Poisson process.

Parameter

`caching` the new value of the caching indicator.

See also `isCaching()`

`public void initCache()`

Resets the random variate generator cache to get the generated inter-arrival times. This method has no effect if caching is disabled.

See also `isCaching()`

`public void clearCache()`

Clears the cached inter-arrival times for this Poisson arrival process. This has some effect only if caching is enabled.

See also `isCaching()`

`public RandomVariateGenWithCache getGenWithCache()`

Returns the random variate generator for the exponential arrival times used when caching is enabled. If caching is disabled, the method throws an `IllegalStateException`.

Returns the random variate generator.

Throws

`IllegalStateException` if caching is disabled.

`public void startStationary()`

This method calls `ContactArrivalProcess.start()` assuming that the λ arrival rate will not change during simulation. Subclasses violating this assumption should override this method.

`public static double[] getMLE (int[] [] arrivals, int numObs, int numPeriods)`

Estimates the parameters of a Poisson arrival process with arrival rate λ from the number of arrivals in the array `arrivals`. Element `arrivals[i][p]` corresponds to the number of arrivals on day `i` during period `p`, where $i = 0, \dots, n - 1$, $p = 0, \dots, P - 1$, $n = \text{numObs}$, and $P = \text{numPeriods}$. This method sums the number of arrivals on every period for each day and uses the resulting array of `numObs` observations to estimate a Poisson arrival rate. This returns an array containing the estimated $\hat{\lambda}$, which estimates the expected number of arrivals during one day.

Parameters

`arrivals` the number of arrivals during each day and period.

`numObs` the number of days.

`numPeriods` the number of periods.

Returns the estimated arrival rates.

```
public static PoissonArrivalProcess getInstanceFromMLE (ContactFactory
                                                         factory,
                                                         RandomStream
                                                         stream, double
                                                         dayLength, int[] []
                                                         arrivals, int
                                                         numObs, int
                                                         numPeriods)
```

Constructs a new arrival process with arrival rate estimated by the maximum likelihood method based on the `numObs` observations in array `arrivals`. Element `arrivals[i][p]` corresponds to the number of arrivals on day `i` during period `p`, where $i = 0, \dots, n - 1$, $p = 0, \dots, P - 1$, $n = \text{numObs}$, and $P = \text{numPeriods}$. The estimated arrival rate, which approximates the expected number of arrivals during a day, is divided by `dayLength` to be relative to one time unit.

Parameters

`factory` the contact factory used to create contacts.

`stream` the random stream to generate arrival times.

`dayLength` the duration of the day, in simulation time units.

`arrivals` the number of arrivals.

`numObs` the number of days.

`numPeriods` the number of periods.

Returns the constructed arrival process.

PiecewiseConstantPoissonArrivalProcess

Represents a non-homogeneous Poisson arrival process with piecewise-constant arrival rates. Each inter-arrival time is an exponential variate with rate $\lambda(t)$, where $\lambda(t) = B\lambda_{p(t)}$ is a piecewise-constant function over simulation time. The function $p(t)$ gives the period corresponding to simulation time t whereas λ_p is the *base arrival rate* for the Poisson process, during period p . This class uses the `PoissonArrivalProcess` base class to generate inter-arrival times and to adjust the arrival time when the rate changes. If a single period p is simulated as if it was infinite in the model, the arrival rate is fixed to λ_p .

```
package umontreal.iro.lecuyer.contactcenters.contact;

public class PiecewiseConstantPoissonArrivalProcess extends
    PoissonArrivalProcess
    implements PeriodChangeListener
```

Field

```
public static double s_bgammaParam
```

Contains the parameter for the gamma-distributed busyness factor given by methods for parameter estimation. This is the alpha parameter of the gamma distribution for busyness. ATTENTION: variable de travail; utiliser tout de suite apres getMLENegMulti

Constructors

```
public PiecewiseConstantPoissonArrivalProcess (PeriodChangeEvent pce,
                                                ContactFactory factory,
                                                double[] lambdas,
                                                RandomStream stream)
```

Constructs a new Poisson arrival process with piecewise-constant arrival rates instantiating new contacts using `factory`. The parameter λ is initialized with `Blambdas[0]`, and is updated at the beginning of each period with a value from `lambdas`. The random number stream `stream` is used to generate the needed uniforms. The newly-constructed arrival process is added to the period-change event `pce` for the arrival rate to be automatically updated.

Parameters

`pce` the period-change event associated with this object.

`factory` the factory instantiating contacts.

`lambdas` the base arrival rates.

`stream` the random number generator for inter-arrival times.

Throws

`IllegalArgumentException` if there is not one rate per period.

`NullPointerException` if any argument is null.

```
public PiecewiseConstantPoissonArrivalProcess (PeriodChangeEvent pce,
                                              ContactFactory factory,
                                              double[] lambdas,
                                              RandomStream stream,
                                              RandomVariateGen bgen)
```

Similar to `(PeriodChangeEvent, ContactFactory, double[], RandomStream)`, but with busyness generator `bgen`. It generates a busyness factor multiplying the base rate.

Parameters

`pce` the period-change event associated with this object.

`factory` the factory instantiating contacts.

`lambdas` the base arrival rates.

`stream` the random number generator for inter-arrival times.

`bgen` random number generator for busyness

Throws

`IllegalArgumentException` if there is not one rate per period.

`NullPointerException` if any argument is null.

Methods

```
public static void setNumMC (int n)
```

Sets the number of Monte Carlo samples to n . This is the number of MC samples used in the `getMLE` method in subclasses.

Parameter

`n`

```
public static int getNumMC()
```

Sets the number of Monte Carlo samples to n . This is the number of MC samples used in the `getMLE` method in subclasses.

```
public boolean isNormalizing()
```

Determines if the base arrival rates are normalized with period duration. When normalization is enabled, for period `p`, the effective base arrival rate is `getLambda (p)/getPeriodChangeEvent().getPeriodDuration (p)`. No normalization is applied for the wrap-up period, because its duration is unknown when it starts. If normalization is disabled (the default), the base arrival rates are used as specified.

Returns if the arrival process normalizes base arrival rates.

```
public void setNormalizing (boolean b)
```

Sets the arrival rates normalization indicator to `b`.

Parameter

`b` the new arrival rate normalization indicator.

See also `isNormalizing()`

```
public PeriodChangeEvent getPeriodChangeEvent()
```

Returns the period-change event associated with this object.

Returns the associated period-change event.

```
public void startStationary()
```

This method checks that the associated period-change event is locked to a fixed period, and calls `ContactArrivalProcess.start()` if this is the case. Otherwise, it throws an `UnsupportedOperationException` since the arrival rate can change with the current period.

```
public double[] getLambdas()
```

Returns the current value of `lambdas`.

Returns the current base rates for the process.

```
public void setLambdas (double[] lambdas)
```

Sets the base arrival rates to `lambdas`.

Parameter

`lambdas` the new base arrival rates.

Throws

`NullPointerException` if the given array is `null`.

`IllegalArgumentException` if the length of the array is smaller than the number of periods.

```
public static double[] getMLE (int[] [] arrivals, int numObs, int
                               numPeriods)
```

Estimates the parameters of a Poisson arrival process with piecewise-constant arrival rate from the number of arrivals in the array `arrivals`, and returns an array giving the estimated arrival rate for each main period. Element `arrivals[i][p]` corresponds to the number of arrivals on day `i` during main period `p`, where $i = 0, \dots, n-1$, $p = 0, \dots, P-1$, $n = \text{numObs}$, and $P = \text{numPeriods}$. This method estimates the expected number of arrivals during main period p , noted λ_p , independently for each main period, assuming that the number of arrivals in that period follows the Poisson distribution. The returned array contains the estimated arrival rate for each of the P periods, noted $\lambda_1, \dots, \lambda_P$.

Parameters

arrivals the number of arrivals during each day and period.

numObs the number of days.

numPeriods the number of periods.

Returns the estimated arrival rates.

```
public static void setVarianceEpsilon (double eps)
```

Sets the lower limit for the variance of the busyness distribution. When the variance would be smaller than eps, the parameter of the busyness distribution is reset so that variance = eps.

```
public static double getVarianceEpsilon()
```

Returns the value of **varianceEpsilon**.

Returns the lower bound of the variance for busyness

```
public static double[] getMLENegMulti (int[] [] arrivals, int numObs, int
                                     numPeriods)
```

Estimates the parameters of a Poisson arrival process with piecewise-constant arrival rate multiplied by a day-specific busyness factor following the $\text{gamma}(\alpha_0, \alpha_0)$ distribution from the number of arrivals in the array **arrivals**. Element **arrivals[i][p]** corresponds to the number of arrivals of this type on day **i** during main period **p**, where $i = 0, \dots, n - 1$, $p = 0, \dots, P - 1$, $n = \text{numObs}$, and $P = \text{numPeriods}$. This returns an array with the estimated arrival rates and stores the gamma busyness parameter in **s.bgammaParam**. This method assumes that the number of arrivals during main periods, represented by the vector A_1, \dots, A_P , follows the negative multinomial distribution with parameters $(\alpha_0, \rho_1, \dots, \rho_P)$ where $\rho_p = \lambda_p / (\alpha_0 + \sum_{k=1}^P \lambda_k)$ for $p = 1, \dots, P$. After $\alpha_0, \rho_1, \dots, \rho_P$ are estimated using maximum likelihood, arrival rate for any main period $p = 1, \dots, P$ can be obtained using $\lambda_p = \alpha_0 \rho_p / \rho_0$, where $\rho_0 = 1 - \sum_{k=1}^P \rho_k$. This method thus returns the array with $\lambda_1, \dots, \lambda_P$.

Parameters

arrivals the number of arrivals during each day and period.

numObs the number of days.

numPeriods the number of periods.

Returns the estimated arrival rates.

```
public static PiecewiseConstantPoissonArrivalProcess getInstanceFromMLE
(PeriodChangeEvent pce, ContactFactory factory, RandomStream stream, int[] []
 arrivals, int numObs, int numPeriods, boolean withGammaBusyness)
```

Constructs a new arrival process with arrival rates estimated by the maximum likelihood method based on the `numObs` observations in array `arrivals`. Element `arrivals[i][p]` corresponds to the number of arrivals on day `i` during period `p`, where $i = 0, \dots, n - 1$, $p = 0, \dots, P - 1$, $n = \text{numObs}$, and $P = \text{numPeriods}$. If `withGammaBusyness` is `true`, the number of arrivals is considered to follow the negative multinomial distribution, and the α_0 parameter for the gamma-distributed busyness factor is stored in `s.bgammaParam`. Otherwise, the periods are considered independent, and the number of arrivals during a period is considered to follow the Poisson distribution. The expected number of arrivals used during the preliminary period is equal to the expectation estimated for the first main period while the arrival rate during the wrap-up period is always 0.

Parameters

`pce` the period-change event marking the end of periods.

`factory` the contact factory used to create contacts.

`stream` the random stream to generate arrival times.

`arrivals` the number of arrivals.

`numObs` the number of days.

`numPeriods` the number of periods.

`withGammaBusyness` determines if the α_0 parameter is estimated in addition to the arrival rates.

Returns the constructed arrival process.

PoissonArrivalProcessWithTimeIntervals

Represents a Poisson arrival process with piecewise-constant arrival rates that can change at arbitrary moments during the simulation. This process is similar to `PiecewiseConstantPoissonArrivalProcess`, except the times arrival rates change do not need to correspond to main periods. More specifically, let $t_0 < \dots < t_L$ be an increasing sequence of simulation times, and let $B\lambda_j$, for $j = 0, \dots, L - 1$, be the arrival rate during time interval $[t_j, t_{j+1})$. The arrival rate is 0 for $t < t_0$ and $t \geq t_L$.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class PoissonArrivalProcessWithTimeIntervals extends
    PoissonArrivalProcess
```

Constructors

```
public PoissonArrivalProcessWithTimeIntervals (ContactFactory factory,
                                                double[] times, double[]
                                                lambdas, RandomStream
                                                stream)
```

Calls `PoissonArrivalProcessWithTimeIntervals (Simulator.getDefaultSimulator(), factory, times, lambdas, stream)`.

```
public PoissonArrivalProcessWithTimeIntervals (Simulator sim,
                                                ContactFactory factory,
                                                double[] times, double[]
                                                lambdas, RandomStream
                                                stream)
```

Constructs a new arrival process using the simulator `sim`, the contact factory `factory` for creating contacts, times t_0, \dots, t_L in array `times`, and arrival rates in array `lambdas`. Inter-arrival times are generated using the random stream `stream`.

Parameters

`sim` the simulator used to schedule events.

`factory` the factory creating contacts for this arrival process.

`times` the sequence of times at which arrival rate changes.

`lambdas` the arrival rates.

`stream` the random stream for inter-arrival times.

Throws

`NullPointerException` if any argument is `null`.

`IllegalArgumentException` if `lambdas.length` is smaller than 1, or if `times.length` does not correspond to `lambdas.length` plus 1, or if `times` is not an increasing sequence of numbers.

Methods

```
public boolean isNormalizing()
```

Determines if the base arrival rates are normalized with length of intervals. When normalization is enabled, for interval $t_{j+1} - t_j$, the effective base arrival rate is $\lambda_j / (t_{j+1} - t_j)$. If normalization is disabled (the default), the base arrival rates are used as specified.

Returns if the arrival process normalizes base arrival rates.

```
public void setNormalizing (boolean b)
```

Sets the arrival rates normalization indicator to `b`.

Parameter

`b` the new arrival rate normalization indicator.

See also `isNormalizing()`

```
public double[] getTimes()
```

Returns the array of times containing t_0, \dots, t_L .

Returns the array of times.

```
public double[] getArrivalRatesInt()
```

Similar to `ContactArrivalProcess.getArrivalRates (double[])`, for the arrival rates per interval.

```
public double[] getExpectedArrivalRatesInt()
```

Similar to `ContactArrivalProcess.getExpectedArrivalRates (double[])`, for the arrival rates per interval.

```
public double[] getExpectedArrivalRatesBInt()
```

Similar to `ContactArrivalProcess.getExpectedArrivalRatesB (double[])`, for the arrival rates per interval.

PoissonGammaArrivalProcess

Represents a doubly-stochastic Poisson process with piecewise-constant randomized arrival rates [11]. The base arrival rates λ_p are constant during each period, but they are not deterministic: for period p , the base rate of the Poisson process is defined as λ_p times a gamma random variable with shape and scale parameters $\alpha_{G,p}$, and mean 1. However, if $\alpha_{G,p}$ or λ_p are 0, the resulting arrival rate during period p is always set to 0. As with the Poisson process with deterministic arrival rates, the generated base arrival rates are multiplied by a global busyness factor B for the day, and also by a busyness factor B_p specific to each period of the day in order to get the arrival rates. Because the values of $\lambda(t)$ are generated once for a replication, in the `init()` method, not calling this method before the simulation starts could lead to unpredictable arrival rates.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class PoissonGammaArrivalProcess extends
    PiecewiseConstantPoissonArrivalProcess
```

Constructor

```
public PoissonGammaArrivalProcess (PeriodChangeEvent pce, ContactFactory
    factory, double[] galphas, double[]
    glambdas, RandomStream stream,
    RandomStream streamBusyness)
```

Constructs a new Poisson-gamma arrival process using `factory` to instantiate contacts. For each period p , the parameters of the gamma rate are given in `galphas[p]` and `glambdas[p]`. The random stream `stream` is used to generate the uniforms for the exponential times whereas the stream `streamBusyness` is used to generate the busyness factors for each period of the day.

Parameters

`pce` the period-change event associated with this object.

`factory` the factory creating contacts for this generator.

`galphas` the $\alpha_{G,p}$ parameters for the gamma variates for busyness.

`glambdas` the λ_p arrival rates.

`stream` random number stream for the exponential variates.

`streamBusyness` random number stream for the gamma rate values.

Throws

`IllegalArgumentException` if there is not one rate for each period.

`NullPointerException` if any argument is `null`.

Methods

`public double[] getGammaAlphas()`

Returns the parameters $\alpha_{G,p}$ of the gamma distribution for busyness.

Returns the $\alpha_{G,p}$ parameters.

`public double[] getGammaLambdas()`

Returns the λ_p parameters for the rates.

Returns the λ_p parameters.

`public void setGammaParams (double[] galphas, double[] glambdas)`

Sets the $\alpha_{G,p}$ and λ_p parameters for the busyness and the arrival rates to `galphas` and `glambdas`, respectively.

Parameters

`galphas` the new $\alpha_{G,p}$ parameters.

`glambdas` the new λ_p rates.

Throws

`NullPointerException` if the given arrays are `null`.

`IllegalArgumentException` if the length of the given arrays does not correspond to at least the number of periods.

`public RandomStream getBusynessStream()`

Returns the random stream used to generate the busyness factors for this arrival process.

Returns the random stream for the values of the busyness factors.

`public void setBusynessStream (RandomStream streamBusyness)`

Changes the random stream used to generate the busyness factors for this arrival process.

Parameter

`streamBusyness` random number stream for the busyness factors.

Throws

`NullPointerException` if the parameter is `null`.

`public static double[] getMLE (int[] [] arrivals, int numObs, int numPeriods)`

Estimates the parameters of a Poisson-gamma arrival process from the number of arrivals in the array `arrivals`. Element `arrivals[i][p]` corresponds to the number of arrivals on day `i` during period `p`, where $i = 0, \dots, n - 1$, and $p = 0, \dots, P - 1$, with $n = \text{numObs}$, and $P = \text{numPeriods}$. This method estimates $\alpha_{G,p}$ and λ_p independently for each period, assuming that the number of arrivals in that period follows the negative binomial distribution with first parameter $\alpha_{G,p}$. The returned array contains $(\alpha_{G,0}, \lambda_0, \dots, \alpha_{G,P-1}, \lambda_{P-1})$.

Parameters

arrivals the number of arrivals during each day and period.

numObs the number of days.

numPeriods the number of periods.

Returns the estimated α_j and λ_j parameters.

```
public static PoissonGammaArrivalProcess getInstanceFromMLE
(PeriodChangeEvent pce, ContactFactory factory, RandomStream stream,
RandomStream streamBusyness, int[][] arrivals, int numObs, int numPeriods)
```

Constructs a new arrival process with gamma arrival rates estimated by the maximum likelihood method based on the **numObs** observations in array **arrivals**. Element **arrivals[i][p]** corresponds to the number of arrivals on day **i** during period **p**, where $i = 0, \dots, n - 1$, $p = 0, \dots, P - 1$, with $n = \text{numObs}$, and $P = \text{numPeriods}$. The parameters of the gamma-distributed arrival rates during the main periods are estimated using **getMLE** (**int[][]**, **int**, **int**). For the preliminary period, the parameters of the first main period are used. For the wrap-up periods, both parameters are set to 0; as a result, the arrival rate is always 0 during the wrap-up period.

Parameters

pce the period-change event marking the end of periods.

factory the contact factory used to create contacts.

stream random stream to generate arrival times.

streamBusyness random stream to generate busyness factors.

arrivals the number of arrivals.

numObs the number of days.

numPeriods the number of periods.

Returns the constructed arrival process.

```
public static double[] getMLEBB (int[][] arrivals, int numObs, int
                                numPeriods, int numMC,
                                ArrivalProcessParams arrPar)
```

Estimates the parameters of a Poisson-gamma arrival process for the case of a global busyness factor for the day, and specific busyness factors for each period of the day, from the number of arrivals in the array **arrivals**. Element **arrivals[i][p]** corresponds to the number of arrivals on day **i** during period **p**, where $i = 0, \dots, n - 1$, and $p = 0, \dots, P - 1$, with $n = \text{numObs}$, and $P = \text{numPeriods}$. This method estimates and returns the parameters of the gamma distribution $\alpha_{G,p}$ and the average rate λ_p for each period. The returned array contains $(\alpha_{G,0}, \lambda_0, \dots, \alpha_{G,P-1}, \lambda_{P-1})$. The global busyness factor is also estimated. This is the case where the arrivals are determined from $B * B_j * \lambda_j$.

Parameters

arrivals the number of arrivals during each day and period.

numObs the number of days.

numPeriods the number of periods.

numMC the number of MonteCarlo samples used in the estimation.

arrPar other parameters of the arrival process.

Returns the estimated gamma and lambda parameters of this process.

PoissonGammaNortaRatesArrivalProcess

Represents a doubly-stochastic Gamma-Poisson process with piecewise-constant randomized correlated arrival rates. The base arrival rates λ_p are constant during each period, but they are not deterministic: for period p , the base rate of the Poisson process is defined as a correlated gamma random variable. The marginal distribution of the rate is gamma with shape parameter $\alpha_{G,p}$, and scale parameter $\lambda_{G,p}$ (mean $\alpha_{G,p}/\lambda_{G,p}$). The correlation structure is modelled using Normal copula model with positive definite correlation matrix Σ having elements in $[-1, 1]$. If $\alpha_{G,p}$ or $\lambda_{G,p}$ are 0, the resulting arrival rate during period p is always 0. As with the Poisson process with deterministic arrival rates, the generated base arrival rates are multiplied by a busyness factor B to get the arrival rates. Because the values of $\lambda(t)$ are generated once for a replication, in the `init()` method, not calling this method before the simulation starts could lead to unpredictable arrival rates.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class PoissonGammaNortaRatesArrivalProcess extends
    PiecewiseConstantPoissonArrivalProcess
```

Constructor

```
public PoissonGammaNortaRatesArrivalProcess (PeriodChangeEvent pce,
                                             ContactFactory factory,
                                             double[] galphas, double[]
                                             glambdas, double[][] CorrMtx,
                                             RandomStream stream,
                                             RandomStream busynessStream)
```

Constructs a new Poisson-gamma arrival process using `factory` to instantiate contacts. For each period p , the parameters of the gamma rate are given in `galphas[p]` and `glambdas[p]`. The random stream `stream` is used to generate the uniforms for the exponential times whereas the stream `busynessStream` is used to generate the gamma rates.

Parameters

`pce` the period-change event associated with this object.

`factory` the factory creating contacts for this generator.

`galphas` the $\alpha_{G,p}$ parameters of the gamma variates.

`glambdas` the $\lambda_{G,p}$ parameters of the gamma variates.

`CorrMtx` the correlation matrix of the Normal copula model for rates.

`stream` random number stream for the exponential variates.

`busynessStream` random number stream for the busyness factor.

Throws

`IllegalArgumentException` if there is not one rate for each period.

`NullPointerException` if any argument is `null`.

Methods

```
public double[] getGammaAlphas()
```

Returns the parameters $\alpha_{G,p}$ for the gamma rates.

Returns the $\alpha_{G,p}$ parameters for this object.

```
public double[] getGammaLambdas()
```

Returns the λ_p parameters for the arrivals rates.

Returns the λ_p parameters.

```
public void setGammaParams (double[] galphas, double[] glambdas)
```

Sets the $\alpha_{G,p}$ and λ_p parameters for the gamma arrival rates to `galphas` and `glambdas`, respectively.

Parameters

`galphas` the new $\alpha_{G,p}$ parameters.

`glambdas` the new λ_p parameters.

Throws

`NullPointerException` if the given arrays are `null`.

`IllegalArgumentException` if the length of the given arrays does not correspond to at least the number of periods.

```
public double[][] getSigma()
```

Returns the correlation matrix associated with this arrival process.

Returns the associated correlation matrix.

```
public void setSigma (double[][] CorrMtx)
```

Sets the associated correlation matrix to `CorrMtx`.

Parameter

`CorrMtx` the new sigma correlation matrix.

Throws

`NullPointerException` if `CorrMtx` is null.

`IllegalArgumentException` if `CorrMtx` is not a $P \times P$ symmetric and positive-definite matrix.

```
public RandomStream getBusynessStream()
```

Returns the random stream used to generate the busyness factors for the Poisson arrival process.

Returns the random stream for the values of λ_p .

```
public void setBusynessStream (RandomStream busynessStream)
```

Changes the random stream used to generate the busyness factors for the Poisson arrival process.

Parameter

`busynessStream` random number generator for the λ_p values.

Throws

`NullPointerException` if the parameter is null.

```
public static double[] getMLE (int[] [] arrivals, int numObs, int
                               numPeriods, int numMC, CorrelationFit fit,
                               double[] [] corr)
```

Estimates the parameters of a Poisson-gamma-norta-rates arrival process from the number of arrivals in the array `arrivals`. Element `arrivals[i][p]` corresponds to the number of arrivals on day i during period p , where $i = 0, \dots, n - 1$, and $p = 0, \dots, P - 1$, with $n = \text{numObs}$, and $P = \text{numPeriods}$. This method estimates and returns the parameters of the gamma distribution $\alpha_{G,p}$ and the average rate λ_p for each period. The returned array contains $(\alpha_{G,0}, \lambda_0, \dots, \alpha_{G,P-1}, \lambda_{P-1})$. It also estimates the correlation matrix using algorithm `fit`, and returns it in `corr`. The memory for the `numPeriods` x `numPeriods` elements of matrix `corr` must be reserved outside this method before calling it.

Parameters

`arrivals` the number of arrivals during each day and period.

`numObs` the number of days.

`numPeriods` the number of periods.

`numMC` the number of MonteCarlo samples used in the estimation.

`fit` type of fit used to compute the correlation matrix.

`corr` the estimated correlation matrix is returned in `corr`.

Returns the estimated gamma and lambda parameters of this process.

```
public static PoissonGammaNortaRatesArrivalProcess getInstanceFromMLE
(PeriodChangeEvent pce, ContactFactory factory, RandomStream stream,
RandomStream busynessStreams, int[] [] arrivals, int numObs, int
numPeriods, int numMC, CorrelationFit fit)
```

Constructs a new arrival process with gamma arrival rates estimated by the maximum likelihood method based on the `numObs` observations in array `arrivals`. Element `arrivals[i][p]` corresponds to the number of arrivals on day `i` during period `p`, where $i = 0, \dots, n - 1$, $p = 0, \dots, P - 1$, $n = \text{numObs}$, and $P = \text{numPeriods}$. The parameters of the gamma-distributed arrival rates during the main periods are estimated using `getMLE (int[] [], int, int, int, CorrelationFit, double[] [])`. For the preliminary period, the parameters of the first main period are used. For the wrap-up periods, both parameters are set to 0; as a result, the arrival rate is always 0 during the wrap-up period.

Parameters

`pce` the period-change event marking the end of periods.

`factory` the contact factory used to create contacts.

`stream` the random stream to generate arrival times.

`busynessStreams` the random stream to generate busyness factors.

`arrivals` the number of arrivals.

`numObs` the number of days.

`numPeriods` the number of periods.

`numMC` the number of MonteCarlo samples used in the estimation.

`fit` type of fit used to compute the correlation matrix.

Returns the constructed arrival process.

GammaParameterEstimator

This class implements the parameter estimation for the doubly Gamma-Poisson process. The rate $T_{i,j}$ of this process consists of three multiplicative contributions: (i) the deterministic piece-wise constant rate λ_i , (ii) the busyness factor for the day, β_j , (iii) the business factor for the sub-period of the day, $B_{i,j}$. The input data are counts observed for I sub-periods of the day during J days. We assume that the rate follows $T_{i,j} = \lambda_i \beta_j B_{i,j}, \forall i = 1 \dots I, j = 1 \dots J$ and input data $Y_{i,j} \sim \text{Poisson}(T_{i,j})$ follow the Poisson distribution conditional on the rates. The busyness factor for the day follows Gamma distribution with parameters Q and Q . The busyness factor follows the Gamma distribution with parameters R and R . The busyness factors are assumed to be independent across days and sub-intervals of the day. The class implements two estimators of the process parameters: the Moment Matching Estimator (MME) in the method `getMMEdoublyGamma()` and the Maximum Likelihood Estimator (MLE) in the method `getMLEdoublyGamma()`. The MME is a suboptimal, but simple and fast estimator based on matching the theoretical means, variances and covariances of the process with their empirical counterparts. The MLE is a statistically optimal estimator with greater accuracy than the MME, but it is less computationally efficient. The MLE is implemented using the stochastic trust-region Gauss-Newton algorithm. The MLE estimator first calls the MME estimator and uses it as a starting point for the optimization. The `startGradUncTrustRegion(double[])` method uses common random numbers to track changes in the cost function and call of this method with option "CostOnly" must, in general, be always preceded by the call of this method without this option in order to provide meaningful results.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class GammaParameterEstimator
```

Constructors

```
public GammaParameterEstimator (int[] [] data, int N, int P)
```

Constructs a new estimator object with a given set of input data

Parameters

data the matrix of input data $Y_{i,j}$ with N rows corresponding to N observations and P columns corresponding to P sub-periods in the day.

N the number of observations

P the number of sub-periods in the day

```
public GammaParameterEstimator (int[] [] data, int N, int P, int M)
```

Constructs a new estimator object with a given set of input data and default seed for the Gamma random variable generators.

Parameters

- data** the matrix of input data $Y_{i,j}$ with N rows corresponding to N observations and P columns corresponding to P sub-periods in the day.
- N** the number of observations
- P** the number of sub-periods in the day
- M** the number of Monte-Carlo samples used in the evaluation of stochastic derivatives and the cost function

```
public GammaParameterEstimator (int[] [] data, int N, int P, int M, long[]
                                Seed)
```

Constructs a new estimator object with a given set of input data and a user defined seed for the Gamma random variable generators.

Parameters

- data** the matrix of input data $Y_{i,j}$ with N rows corresponding to N observations and P columns corresponding to P sub-periods in the day.
- N** the number of observations
- P** the number of sub-periods in the day
- M** the number of Monte-Carlo samples used in the evaluation of stochastic derivatives and the cost function

Seed is the vector of 6 integers. The first 3 values of the seed must all be less than $m_1 = 4294967087$, and not all 0; and the last 3 values must all be less than $m_2 = 4294944443$, and not all 0.

Methods

```
public double[] getMMEdoublyGamma()
```

Estimates the parameters of a doubly Gamma Poisson-Gamma arrival process that has both busyness factor for the day and the busyness factor for the sub-period of the day, both following the Gamma distribution, from the number of arrivals in the array **arrivals** using method of moments. The day-specific busyness factor follows the $\text{Gamma}(Q, Q)$ distribution, the sub-period-specific busyness factor follows the $\text{gamma}(R, R)$ distribution. Element **arrivals**[*i*][*p*] corresponds to the number of arrivals on day *i* during period *p*, where $i = 0, \dots, n-1$, and $p = 0, \dots, P-1$, $n = \text{numObs}$, and $P = \text{numPeriods}$. If we follow the notation introduced for **PoissonGammaArrivalProcess**, then this method estimates $\alpha_{G,p}$, $\lambda_{G,p}$ and the daily gamma busyness parameter. It is assumed that $\alpha_{G,p}$ is a vector of distinct values while all the entries of $\lambda_{G,p}$ are the same and equal to R . The estimation is based on matching the empirical first and second order moments of the distribution of counts (mean, variance and covariance) with the analytical moments of the doubly Gamma Poisson-Gamma arrival process distribution. The returned array of $2P+1$ elements contains $(\alpha_{G,0}, \lambda_{G,0}, \dots, \alpha_{G,P-1}, \lambda_{G,P-1}), \beta_0$.

Returns the estimated gamma parameters.

```
public double[] getMMEdoublyGammaGeneral()
```

Estimates the parameters of a doubly Gamma Poisson-Gamma arrival process that has both busyness factor for the day and the busyness factor for the sub-period of the day, both following the Gamma distribution, from the number of arrivals in the array `arrivals` using method of moments. The day-specific busyness factor follows the $\text{Gamma}(Q, Q)$ distribution, the sub-period-specific busyness factor follows the $\text{gamma}(R, R)$ distribution. Element `arrivals[i][p]` corresponds to the number of arrivals on day i during period p , where $i = 0, \dots, n-1$, and $p = 0, \dots, P-1$, $n = \text{numObs}$, and $P = \text{numPeriods}$. If we follow the notation introduced for `PoissonGammaArrivalProcess`, then this method estimates $\alpha_{G,p}$, $\lambda_{G,p}$ and the daily gamma busyness parameter. It is assumed that $\alpha_{G,p}$ and $\lambda_{G,p}$ are vectors of distinct values. The estimation is based on matching the empirical first and second order moments of the distribution of counts (mean, variance and covariance) with the analytical moments of the doubly Gamma Poisson-Gamma arrival process distribution. The returned array of $2P + 1$ elements contains $(\alpha_{G,0}, \lambda_{G,0}, \dots, \alpha_{G,P-1}, \lambda_{G,P-1}), \beta_0$.

Returns the estimated gamma parameters.

```
public double[] getMLEdoublyGamma()
```

Estimates the parameters of a doubly Gamma Poisson-Gamma arrival process that has both busyness factor for the day and the busyness factor for the sub-period of the day, both following the Gamma distribution, from the number of arrivals in the array `arrivals` using maximum likelihood approach. It uses the trust region Gauss-Newton optimizer implemented in `startGradUncTrustRegion (double[])` and the stochastic approximation of the likelihood function and its derivatives implemented in `getLikelihoodDerivatives-DoublyGamma (double[], double, double, String)` in order to obtain the MLE. Before launching the Gauss-Newton optimizer, this algorithm calls the `getMMEdoublyGamma()` in order to get good initialization for the values of parameters. The output is formatted the same way as it is done for the `getMMEdoublyGamma()`. If we follow the notation introduced for `PoissonGammaArrivalProcess`, then this method estimates $\alpha_{G,p}$, $\lambda_{G,p}$ and the daily gamma busyness parameter. It is assumed that $\alpha_{G,p}$ is a vector of distinct values while all the entries of $\lambda_{G,p}$ are the same and equal to R . Thus the returned array of $2P + 1$ elements contains $(\alpha_{G,0}, \dots, \alpha_{G,P-1}, \lambda_{G,0}, \dots, \lambda_{G,P-1}, \beta_0)$.

Returns the estimated gamma parameters.

```
public double[] getMLEdoublyGammaSpline()
```

Estimates the parameters of a doubly Gamma Poisson-Gamma arrival process that has both busyness factor for the day and the busyness factor for the sub-period of the day, both following the Gamma distribution, from the number of arrivals in the array `arrivals` using maximum likelihood approach. It uses the trust region Gauss-Newton optimizer implemented in `startGradUncTrustRegion (double[])` and the stochastic approximation of the likelihood function and its derivatives implemented in `getLikelihoodDerivatives-DoublyGamma (double[], double, double, String)` in order to obtain the MLE. Before launching the Gauss-Newton optimizer, this algorithm calls the `getMMEdoublyGamma()` in order to get good initialization for the values of parameters. The output is formatted the same

way as it is done for the `getMMEdoublyGamma()`. If we follow the notation introduced for `PoissonGammaArrivalProcess`, then this method estimates $\alpha_{G,p}$, $\lambda_{G,p}$ and the daily gamma busyness parameter. It is assumed that $\alpha_{G,p}$ is a vector of distinct values while all the entries of $\lambda_{G,p}$ are the same and equal to R . Thus the returned array of $2P + 1$ elements contains $(\alpha_{G,0}, \dots, \alpha_{G,P-1}, \lambda_{G,0}, \dots, \lambda_{G,P-1}, \beta_0)$.

Returns the estimated gamma parameters.

```
public double[] getLikelihoodDerivativesDoublyGamma (double[] Lam, double
                                                    R, double Q, String
                                                    OutType)
```

Calculates the values of the log-likelihood function and its derivatives for the doubly Gamma-Poisson arrival process model. Element `arrivals[i][p]` corresponds to the number of arrivals on day i during period p , where $i = 0, \dots, I - 1$, and $p = 0, \dots, P - 1$, $I = \text{numObs}$, and $P = \text{numPeriods}$.

For the doubly Gamma-Poisson arrival process model the log-likelihood function does not admit any closed form as it contains an integral over β_j , the daily busyness factor. The integral cannot be treated analytically. This integral is thus treated numerically via the Monte-Carlo approach. The Monte-Carlo approach uses `numSamples` samples for the evaluation of the integral. The Gamma distribution with both parameters equal to Q is taken as the proposal distribution. The cost function and all the derivatives are thus approximated stochastically. There is an option to return only the value of the cost function (the log-likelihood function) by assigning to `OutType` the string "CostOnly". In this case the algorithm uses the random numbers common with the ones that were used to evaluate the derivatives last time the function was called. The importance sampling is used to compensate for possible change of integration measure (that may arise due to change in the value of Q between calls). This feature is used by the trust region algorithm for determining the quality of the trust region and for regulating the step size of the optimization. The common random numbers approach reduces the effects of noise on the step size regulation.

The returned array of $2 * (P + 2) + 1$ elements contains

- 1) The value of the cost function
- 2) P values of first order derivatives for deterministic rates
- 3) Derivative with respect to $\$R\$$, sub-period Gamma rate
- 4) Derivative with respect to $\$Q\$$, daily Gamma rate
- 5) P values of second order derivatives for deterministic rates
- 6) Second order derivative with respect to $\$R\$$, sub-period Gamma rate
- 7) Second order derivative with respect to $\$Q\$$, daily Gamma rate

Parameters

`Lam` initial values of base rates.

`R` initial value of the Gamma distribution parameter for the sub-period of the day business factor.

`Q` initial value of the Gamma distribution parameter for the daily business factor.

`OutType` the string controlling output options

Returns the values of the log-likelihood function and its first and second order derivatives.

```
public double[] [] getLikelihoodDerivativesDoublyGammaSpline
(double[] Lam, double[] R, double Q, String OutType)
```

Calculates the values of the log-likelihood function and its derivatives for the doubly Gamma-Poisson arrival process model. Element `arrivals[i][p]` corresponds to the number of arrivals on day `i` during period `p`, where $i = 0, \dots, I - 1$, and $p = 0, \dots, P - 1$, $I = \text{numObs}$, and $P = \text{numPeriods}$.

For the doubly Gamma-Poisson arrival process model the log-likelihood function does not admit any closed form as it contains an integral over β_j , the daily busyness factor. The integral cannot be treated analytically. This integral is thus treated numerically via the Monte-Carlo approach. The Monte-Carlo approach uses `numSamples` samples for the evaluation of the integral. The Gamma distribution with both parameters equal to Q is taken as the proposal distribution. The cost function and all the derivatives are thus approximated stochastically. There is an option to return only the value of the cost function (the log-likelihood function) by assigning to `OutType` the string "CostOnly". In this case the algorithm uses the random numbers common with the ones that were used to evaluate the derivatives last time the function was called. The importance sampling is used to compensate for possible change of integration measure (that may arise due to change in the value of Q between calls). This feature is used by the trust region algorithm for determining the quality of the trust region and for regulating the step size of the optimization. The common random numbers approach reduces the effects of noise on the step size regulation.

The returned 2D array Output of $P \times 7$ elements contains

- 1) Output[0][0] The value of the cost function
- 2) Output[:,1] (second column) P values of first order derivatives with respect to deterministic rates
- 3) Output[:,2] P values of second order derivatives with respect to deterministic rates
- 4) Output[:,3] P values of first order derivatives with respect to R_p , sub-period Gamma rate
- 5) Output[:,4] P values of second order derivatives with respect to R_p , sub-period Gamma rate
- 6) Output[0][5] Derivative with respect to Q , daily Gamma rate
- 7) Output[0][6] Second order derivative with respect to Q , daily Gamma rate

Parameters

Lam initial values of base rates.

R initial value of the Gamma distribution parameter for the sub-period of the day business factor.

Q initial value of the Gamma distribution parameter for the daily business factor.

OutType the string controlling output options

Returns the 2D array containing values of the log-likelihood function and its first and second order derivatives.

```
public double[] startGradUncTrustRegionSpline (double[] SolInit)
```

Implements the trust region Gauss-Newton maximizer of the likelihood of the doubly Gamma-Poisson process. In this algorithm the expressions for the second order derivatives (or their approximations having reduced number of terms) are used as pre-scalers for the gradients. Moreover, there is also a set of step size regulators that reflect the size of the trust region, the region in which we believe our model of the cost function (the locally linear model based on the first order Taylor expansion of the cost function) is correct. Step size regulators determine the magnitude of the step at each optimization iteration. We have three step size regulators: for the rates λ_i , for R and for Q . By choosing three different step size regulators for each group of parameters we make sure that we can choose optimal gradient scaling for each group of parameters. This compensates for the fact that the second order derivatives (or their approximations) provide suboptimal scaling of the gradient by either over- or underestimating the necessary scale of gradient step. The factor by which the scaling is under- or overestimated is typically different for different groups of parameters. Finally, because our optimization uses stochastic approximation of derivatives, there is some noise in the gradients. To account for this fact we two methods could be used. First option is the third scaling factor, the noise attenuator **eta**, which is a positive constant smaller than one. In the case **eta** is smaller than 1, only a portion of the gradient is used during each iteration. As the number of Monte-Carlo samples used to approximate the derivatives reduces, the (absolute) value of the noise attenuator must also be reduced. The second option is to use the stochastic approximation approach with the step size reduction sequence of the form $t^{-\alpha}$, where t is the iteration number and α is the number between 1/2 and 1. In our implementation we have parameter **pwr** with default value 5/6 that determines the speed of decay of the step sequence and is equivalent to α .

Parameter

SolInit the value of the solution at first iteration

Returns xiter the MLE estimator of the parameters of the doubly Gamma-Poisson process

```
public void setTrustRegionMaxIterations (int Value)
```

Sets the maximum number of iterations in the trust region optimization algorithm

Parameter

Value maximum number of iterations in the trust region optimization algorithm

```
public void setTrustRegionQualBounds (double LowQualBound, double
                                     HighQualBound)
```

Sets the lower and higher bounds on the quality metrics of the trust region. If the quality metric has value lower than **c0** then the size of the trust region (the step size regulator) will be reduced. If the quality metric has value higher than **c1** then the size of the trust region (the step size regulator) will be increased.

Parameters

`LowQualBound` is the lower bound

`HighQualBound` is the higher bound

```
public void setTrustRegionMultipliers (double IncreaseMultiplier, double  
                                       DecreaseMultiplier)
```

Sets the multipliers that control the rate at which the step size regulator is increased (`g1`) or decreased (`g0`). If the quality metric has value lower than `c0` then the size of the trust region (the step size regulator) is multiplied by `g0`. If the quality metric has value higher than `c1` then the size of the trust region (the step size regulator) will be multiplied by `g1`. `IncreaseMultiplier` must be greater than 1, `DecreaseMultiplier` must be between 0 and 1.

Parameters

`IncreaseMultiplier`

`DecreaseMultiplier`

```
public void setTrustRegionTol (double Value)
```

Sets the threshold for the tolerance of the trust region optimization algorithm. If the difference between the updated and the previous values of the cost function is greater than this value, algorithm stops and returns current values of parameters as a solution.

Parameter

`Value` the value of the stopping criterion

```
public void setTrustRegionNoiseAttenuator (double Value)
```

Sets the value of the noise attenuator for the trust region approach. The optimization algorithm is based on stochastic derivatives. Because of this derivatives contain noise and to reduce its adverse effects we use only portion of the gradient to during each update. As the number of Monte-Carlo samples used to approximate the derivatives reduces, the (absolute) value of the noise attenuator must also be reduced.

Parameter

`Value` the value of the noise attenuator

```
public void setTrustRegionInitBoundary (double Value)
```

Sets the initial size of the trust region (initial value of the step size regulator). This value should be reasonably small to prevent the algorithm from making unreasonably large initial steps at the beginning, when the optimal scaling for the gradients is not known. As the optimization progresses, the step size regulator grows (if necessary) fast at the geometric rate and the optimal scaling for the derivatives is quickly learnt.

Parameter

`Value` the initial value of the step size regulator.

```
public void setTrustRegionAnnealingPwr (double Value)
```

Sets the power law in the step size stochastic approximation annealing sequence.

Parameter

Value

```
public void setSmoothingLambda (double lambda)
```

Sets the smoothing parameter for the use with the smoothing spline doubly gamma penalized MLE.

Parameter

`lambda` smoothing parameter in the interval $[0, 1]$. When this parameter is equal to 1, the smoothing spline is not used and shape parameters are assumed to be independent across sub-periods. Default value 0.95.

```
public void setMovingWindowSize (int movWindowSize)
```

Sets the number of sub-periods over which to average the MME estimate in `getMMEdoublyGammaGeneral`. Without averaging the MME estimates are too noisy. If we want no averaging we can set `movWindowSize=1`. Default value 5.

Parameter

`movWindowSize` number of sub-periods over which to average the MME estimate.

```
public double[][] getOptimizationTrace()
```

Returns the the matrix of the optimization trace containing the evolution of parameters during optimization iterations.

```
public double[] getPolyakAverage()
```

Calculates the estimates of the parameters based on the stochastic optimization framework described by B T Polyak and A B Juditsky in "Acceleration of Stochastic Approximation by Averaging", SIAM J Control Optim. 30(4) 838?855. For the proper use of this option the trust region optimization algorithm must be configured correspondingly. In particular, recommended settings for parameters are `Rinit` 0.1, `eta` 0.3 and 5/6.

Returns The average of the optimization trace over iterations.

```
public void estimateNortaRateParamsStochasticRootFinding  
( )
```

Estimates the parameters of the Gamma-Poisson NORTA model for rates using stochastic root finding approach. It stores the estimated Gaussian copula correlation matrix in `yGauss-Corr`, the estimated base rates in `Qout` and the parameter of the Gamma distribution in `LamOut`. These quantities can be accessed using methods `getNortaRateGaussCorr()`, `getNortaRateGammaShape()` and `getNortaRateGammaScale()`. The parameters of the Gamma distribution are estimated using method `getNegBinMLE (int[], double)`. The entries of the copula correlation matrix are estimated using method `getNortaRhoStochasticRootFinding (double, double[][], double)`.

```
public double[] getNegBinMLE (int[] X, double tole)
```

Calculates the MLEs of parameters of the negative binomial distribution.

Parameters

X vector of observed counts

tole parameter search tolerance for the binary search

Returns Array of size 2 containing MLEs of the parameters of the negative binomial distribution. First element of the array is the MLE of the number of failures. Second element of the returned array is the MLE of the success probability.

```
public double getLogNegBinDer (int[] X, double Xmean, double r)
```

Calculates the derivative of the log-likelihood function for the Negative binomial distribution.

Parameters

X data values

Xmean mean of the data values

r parameter of the negative binomial distribution (number of failures)

Returns derivative of the log-likelihood function for the Negative binomial distribution with respect to parameter r.

```
public double getNortaRhoStochasticRootFinding (double rhoTarget, double[] []  
                                                NegBinParams, double  
                                                rhoInit)
```

Solves the problem of fitting the NORTA correlation coefficient to the empirical Spearman correlation coefficient of counts in the Gamma-Poisson copula model using stochastic root finding approach.

Parameters

rhoTarget the empirically observed Spearman correlation coefficient of counts in the Gamma-Poisson copula model

NegBinParams estimated parameters of the marginal distribution, which is Negative Binomial in the case of Gamma-Poisson copula model

rhoInit initial value of the NORTA correlation coefficient. Typically, rhoInit = rhoTarget

Returns Fitted NORTA correlation coefficient

```
public double[] getNortaRateGammaShape()
```

Returns the estimated vector of α_G parameters of the Gamma distribution in the compound Gamma-Poisson NORTA model for rates. The definition of vector α_G follows that introduced in `PiecewiseConstantPoissonArrivalProcess` so that the base rate in subperiod p is equal to $\alpha_{G,p}/\lambda_{G,p}$.

Returns the estimated vector of α_G

```
public double[] getNortaRateGammaScale()
```

Returns the estimated vector of λ_G parameters of the Gamma distribution in the compound Gamma-Poisson NORTA model for rates. The definition of vector λ_G follows that introduced in `PiecewiseConstantPoissonArrivalProcess` so that the base rate in subperiod p is equal to $\alpha_{G,p}/\lambda_{G,p}$.

Returns the estimated vector of λ_G

```
public double[][] getNortaRateGaussCorr()
```

Returns the estimated copula correlation matrix for the Gamma-Poisson NORTA model for rates.

Returns the estimated copula correlation matrix

```
public double[][] getNortaRateGaussCorrCorrected()
```

Returns the estimated and corrected copula correlation matrix for the Gamma-Poisson NORTA model for rates. The estimated correlation matrix, which is not necessarily positive definite is transformed into a positive definite matrix using the POSDEF algorithm described in [6].

Returns the estimated and corrected copula correlation matrix

```
public double getNortaRateGaussCorrFitMarkovSingleRho()
```

Fits the single ρ Markov linear model $r_j = b^j$ to the estimated copula correlation matrix for the Gamma-Poisson NORTA model for rates.

Returns b

```
public double[] getNortaRateGaussCorrFitGeneralLinear()
```

Fits the general linear model $r_j = ab^j + c$ to the estimated copula correlation matrix for the Gamma-Poisson NORTA model for rates. Returns a vector of length 3 with parameters a , b and c .

Returns vector $[a, b \text{ and } c]$

DirichletCompoundArrivalProcess

Represents a generalization of the non-homogeneous Poisson process where the arrival rates are generated from a Dirichlet compound negative multinomial distribution [16]. As proven in [3], if the arrival rate of a Poisson process is a piecewise-constant function of the simulation time given by $\lambda(t) = B\lambda_{p(t)}$, B being a gamma-distributed busyness factor with shape parameter γ , the distribution of the vector (A_1, \dots, A_P) giving the number of arrivals in each main period is the negative multinomial with parameters $(\gamma, \rho_1, \dots, \rho_{P+1})$ [10, page 292], where

$$\rho_p = \frac{\lambda_p}{1 + \sum_{j=1}^P \lambda_j}, \quad (1)$$

for $p = 1, \dots, P$, and $\rho_{P+1} = 1 - \sum_{j=1}^P \rho_j$.

This arrival process generalizes the previous process by modeling $\mathbf{A} = (A_1, \dots, A_P)$ with a Dirichlet compound negative multinomial distribution [16] instead of a negative multinomial. In this model, the user specifies γ as well as $\alpha_1, \dots, \alpha_{P+1}$. At the beginning of each replication, when base arrival rates are needed, the vector $(\rho_1, \dots, \rho_{P+1})$ is generated from the Dirichlet distribution with parameters $(\alpha_1, \dots, \alpha_{P+1})$, and the base arrival rates $\lambda_1, \dots, \lambda_P$ are determined by solving (1). This results in $\lambda_p = \rho_p / \rho_{P+1}$. During preliminary and wrap-up periods, the base arrival rate is set to 0.

The inter-arrival times are generated using the rates $B\lambda_p$, where B is a busyness given by the user. Note that this variability factor should be gamma-distributed with shape parameter γ and scale parameter 1 to remain consistent with the Dirichlet compound model.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class DirichletCompoundArrivalProcess extends
    PiecewiseConstantPoissonArrivalProcess
```

Constructor

```
public DirichletCompoundArrivalProcess (PeriodChangeEvent pce,
                                         ContactFactory factory, double[]
                                         alphas, RandomStream stream,
                                         RandomStream streamRates)
```

Constructs a new Dirichlet compound Poisson arrival process. The constructed process uses the period-change event `pce`, creates contacts using the factory `factory`, and uses the Dirichlet parameters `alphas`. The random stream `stream` is used for the uniforms for inter-arrival times, and `streamRates` is used for Dirichlet.

Parameters

pce the period-change event associated with this object.

factory the factory creating contacts for this generator.

alphas the values of the α_p parameters.

stream the random number stream for the exponential variates.

streamRates the random number stream for the Dirichlet compound arrival rates.

Throws

IllegalArgumentException if the number of main periods is not `alphas.length - 1`, or if one α_p value is negative or 0.

NullPointerException if any argument is `null`.

Methods

```
public double getAlpha (int p)
```

Returns the value of the α_p parameter for the Dirichlet distribution.

Parameter

p the index of the parameter.

Returns the value of the parameter.

```
public void setAlphas (double[] alphas)
```

Sets the Dirichlet parameters α_p for this object.

Parameter

alphas a new vector of parameters.

Throws

IllegalArgumentException if the length of **alphas** is smaller than $P + 1$, where P is the number of main periods, or if one or more α_p values are negative or 0.

```
public RandomStream getRateStream()
```

Returns the random stream used to generate the rates for the Poisson arrival process.

Returns the random stream for the values of λ_p .

```
public void setRateStream (RandomStream streamRates)
```

Changes the random stream used to generate the rates for the Poisson arrival process to **streamRates**.

Parameter

streamRates the random number generator for the λ_p values.

Throws

`NullPointerException` if the parameter is `null`.

```
public static double[] getMLE (int[] [] arrivals, int numObs, int
                               numPeriods)
```

Estimates the parameters of a Dirichlet compound negative multinomial arrival process with a busyness factor following the $\text{gamma}(\gamma, 1)$ distribution from the number of arrivals in the array `arrivals`. Element `arrivals[i][p]` corresponds to the number of arrivals on day `i` during period `p`, where $i = 0, \dots, n - 1$, and $p = 0, \dots, P - 1$, $n = \text{numObs}$, and $P = \text{numPeriods}$. This returns the α_p Dirichlet parameters, for $P = 0, \dots, P$, and stores the gamma busyness parameter in `PiecewiseConstantPoissonArrivalProcess.s_bgammaParam`.

Parameters

`arrivals` the number of arrivals during each day and period.

`numObs` the number of days.

`numPeriods` the number of periods.

Returns the estimated Dirichlet parameters.

```
public static DirichletCompoundArrivalProcess getInstanceFromMLE
(PeriodChangeEvent pce, ContactFactory factory, RandomStream stream,
 RandomStream streamRates, int[] [] arrivals, int numObs, int numPeriods)
```

Constructs a new arrival process with parameters estimated by the maximum likelihood method based on the `numObs` observations in array `arrivals`. Element `arrivals[i][p]` corresponds to the number of arrivals on day `i` during period `p`, where $i = 0, \dots, n - 1$, $p = 0, \dots, P - 1$, $n = \text{numObs}$, and $P = \text{numPeriods}$. The number of arrivals is considered to follow the Dirichlet compound negative multinomial distribution, and the γ parameter for the gamma busyness factor is stored in `PiecewiseConstantPoissonArrivalProcess.s_bgammaParam`.

Parameters

`pce` the period-change event marking the end of periods.

`factory` the contact factory used to create contacts.

`stream` the random stream to generate arrival times.

`streamRates` the random stream to generate Dirichlet vectors from.

`arrivals` the number of arrivals.

`numObs` the number of days.

`numPeriods` the number of periods.

Returns the constructed arrival process.

PoissonUniformArrivalProcess

This arrival process can be used when the number of arrivals per period A_p is known (when $B = 1$). By default, for each period p , $A_p^* = \text{round}(BA_p)$ uniforms are generated and sorted in increasing order to get the inter-arrival times, supposing we have a Poisson process with stationary increments. Because this algorithm requires the duration of each period, arrivals are not allowed during the wrap-up period, which has a random duration not known at the time arrivals are generated. The algorithm for generating arrival times can be customized by overriding `computeArrivalTimes()`. The number of arrivals, constant by default, can also be changed between replications.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class PoissonUniformArrivalProcess extends ContactArrivalProcess
```

Field

```
protected DoubleArrayList times
```

Array list containing the arrival times of contacts.

Constructor

```
public PoissonUniformArrivalProcess (PeriodChangeEvent pce, ContactFactory
                                     factory, int[] arrivals, RandomStream
                                     stream)
```

Constructs a new arrival process with known number of arrivals in each period. The constructed process uses period-change event `pce`, contact factory `factory`, mean number of arrivals `arrivals[p]` in period `p`, and random number stream `stream` to generate random values.

Parameters

`pce` the period-change event defining the periods.

`factory` the contact factory used to create contacts.

`arrivals` the mean number of arrivals in each period.

`stream` the random number stream for the uniforms.

Throws

`IllegalArgumentException` if there is not a mean number of arrivals for each period.

`NullPointerException` if any argument is `null`.

Methods

```
public RandomStream getStream()
```

Returns the random number stream used to generate uniforms.

Returns the associated random number stream.

```
public void setStream (RandomStream stream)
```

Sets the random number stream to `stream` for generating uniforms.

Parameter

`stream` the new random number stream.

Throws

`NullPointerException` if `stream` is null.

```
public PeriodChangeEvent getPeriodChangeEvent()
```

Returns the period-change event associated with this object.

Returns the associated period-change event.

```
public double getArrivalRate (double st, double et)
```

Computes the arrival rate using the period-change event returned by `getPeriodChangeEvent()` to determine the boundaries of periods, and the arrival rates returned by `getArrivalRate (int)`.

```
public double getExpectedArrivalRate (double st, double et)
```

Computes the expected arrival rate using the period-change event returned by `getPeriodChangeEvent()` to determine the boundaries of periods, and the expected arrival rates returned by `getExpectedArrivalRate (int)`.

```
public int[] getArrivals()
```

Returns the number of arrivals for each period.

Returns the array of number of arrivals.

```
public void setArrivals (int[] arrivals)
```

Sets the number of arrivals in each period to `arrivals`.

Parameter

`arrivals` the number of arrivals.

Throws

`NullPointerException` if the array is `null`.

`IllegalArgumentException` if the length of the given array does not correspond to the number of periods as defined by `getPeriodChangeEvent()`.

`protected void computeArrivalTimes()`

This is called by `init()` to compute the arrival times based on the number of arrivals in each period. The arrival times must be stored in the array list `times`, which is empty at the time the method is called. The arrival times in the list are assumed to be sorted in increasing order after the method returns. The method should use the random stream returned by `getStream()` to generate the random numbers.

By default, for each period $p = 0, \dots, P$ (preliminary and main periods), this generates A_p^* uniforms in $[t_{p-1}, t_p)$, where $t_{-1} = 0$, A_p is the expected number of arrivals in period p , and t_p is the ending time of period p . The generated arrival times are then sorted. If $B = 1$, $A_p^* = A_p$, otherwise $A_p^* = \text{round}(A_p B)$.

FixedCountsArrivalProcess

Represents an arrival process in which the numbers of arrivals per-period C_p (the counts) are given (in a file or directly). A_0 and A_{P+1} , the number of arrivals during the preliminary and the wrap-up periods, respectively, are always 0 for this process. The busyness factor is always 1.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class FixedCountsArrivalProcess extends PoissonUniformArrivalProcess
```

Constructor

```
public FixedCountsArrivalProcess (PeriodChangeEvent pce, ContactFactory
                                factory, int[] counts, RandomStream
                                stream)
```

Constructs a new Poisson arrival process conditional on the number of arrivals being given in each period. With period-change event **pce**, contact factory **factory**, number of arrivals in each period **arrivals**, and random number stream **stream**.

Parameters

pce the period-change event defining the periods.

factory the contact factory instantiating contacts.

counts the number of arrivals in each period.

stream the random number stream for uniform arrival times.

Throws

NullPointerException if one argument is null.

IllegalArgumentException if the length of **arrivals** do not correspond to number of main periods P .

DirichletArrivalProcess

Represents an arrival process where the number of arrivals are spread in periods using a Dirichlet distribution [3]. Let's define the vector of ratios

$$\mathcal{Q} \equiv (\mathcal{Q}_1, \dots, \mathcal{Q}_P) = (A_1/A, \dots, A_P/A),$$

where A_p denotes the number of arrivals during main period p and

$$A = \sum_{p=1}^P A_p$$

is the total number of arrivals. The number of arrivals during the preliminary and the wrap-up periods, A_0 and A_{P+1} respectively, are always 0 for this process.

At the beginning of each replication, A is generated from a probability distribution such as gamma. A vector \mathcal{Q} is then generated from a Dirichlet distribution [10] with parameters $(\alpha_1, \dots, \alpha_P)$. Each component of \mathcal{Q} is multiplied with A to get $\tilde{\mathbf{A}}$ before the vector \mathbf{A} is obtained by rounding each component of $\tilde{\mathbf{A}}$ to the nearest integer.

Since per-period numbers of arrivals are generated directly rather than through arrival rates, this process does not arise as a Poisson arrival process. However, inter-arrival times are generated as if the $A_p^* = \text{round}(BA_p)$ were Poisson variates. As a result, for each main period, the arrival process generates A_p^* uniforms ranging from the beginning to the end of the period, and the uniforms are sorted to get inter-arrival times.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class DirichletArrivalProcess extends PoissonUniformArrivalProcess
```

Constructor

```
public DirichletArrivalProcess (PeriodChangeEvent pce, ContactFactory
                                factory, double[] alphas, RandomStream
                                stream, RandomVariateGen agen)
```

Constructs a new Dirichlet arrival process with period-change event **pce**, contact factory **factory**, Dirichlet parameters **alphas**, random number stream **stream**, and generator **agen** for the number of arrivals.

Parameters

pce the period change event.

factory the contact factory instantiating contacts.

alphas the parameters of the Dirichlet distribution.

stream the random number stream for Dirichlet vectors and uniform arrival times.

agen the random variate generator for the number of arrivals.

Throws

`IllegalArgumentException` if there is not an α value for each main period, or if one α value is negative or 0.

`NullPointerException` if one argument is `null`.

Methods

```
public RandomVariateGen getNumArrivalsGenerator()
```

Returns the random variate generator used for the total number of arrivals A .

Returns the random variate generator for the total number of arrivals.

```
public void setNumArrivalsGenerator (RandomVariateGen agen)
```

Changes the random variate generator for the number of arrivals to `agen`.

Parameter

`agen` the new random variate generator for the number of arrivals.

Throws

`NullPointerException` if the parameter is `null`.

```
public double getAlpha (int p)
```

Returns the value of the α_p parameter for the Dirichlet distribution.

Parameter

`p` the index of the parameter.

Returns the value of the parameter.

```
public void setAlphas (double[] alphas)
```

Sets the Dirichlet parameters α_p for this object.

Parameter

`alphas` a new vector of parameters.

Throws

`IllegalArgumentException` if the length of `alphas` does not correspond to the number of main periods or if one of the α parameter is negative or 0.

`NullPointerException` if `alphas` is `null`.

```
public void initWithFixedA (double a)
```

Initializes the number of arrivals with a fixed A `a`.

Parameter

`a` the total number of arrivals.

Throws

`IllegalArgumentException` if `a` is negative or 0.

```
public static double[] getMLE (int[] [] arrivals, int numObs, int
                               numPeriods)
```

Estimates the Dirichlet parameters of an arrival process from the number of arrivals in the array `arrivals`. Element `arrivals[i][p]` corresponds to the number of arrivals on day `i` during period `p`, where $i = 0, \dots, n - 1$, and $p = 0, \dots, P - 1$, $n = \text{numObs}$, and $P = \text{numPeriods}$. This method computes $\rho_{i,p} = X_{i,p}/Y_i$ where $X_{i,p}$ is the number of arrivals on day i during period p , and Y_i is the total number of arrivals during day i . The returned array contains the Dirichlet parameters $\hat{\alpha}_0, \dots, \hat{\alpha}_{P-1}$ estimated by assuming that the ratios $\rho_{i,p}$ follow the Dirichlet distribution.

Parameters

`arrivals` the number of arrivals during each day and period.

`numObs` the number of days.

`numPeriods` the number of periods.

Returns the estimated Dirichlet parameters.

```
public static DirichletArrivalProcess getInstanceFromMLE
(PeriodChangeEvent pce, ContactFactory factory, RandomStream stream,
 RandomVariateGen agen, int[] [] arrivals, int numObs, int numPeriods)
```

Constructs a new arrival process with Dirichlet parameters estimated by the maximum likelihood method based on the `numObs` observations in array `arrivals`. Element `arrivals[i][p]` corresponds to the number of arrivals on day `i` during period `p`, where $i = 0, \dots, n - 1$, $p = 0, \dots, P - 1$, $n = \text{numObs}$, and $P = \text{numPeriods}$. The created arrival process uses the random variate generator `agen` to generate the total number of arrivals for each day while the Dirichlet parameters are estimated using `getMLE (int[] [], int, int)`.

Parameters

`pce` the period-change event marking the end of periods.

`factory` the contact factory used to create contacts.

`stream` the random stream to generate arrival times.

`agen` the random variate generator for A .

`arrivals` the number of arrivals.

`numObs` the number of days.

`numPeriods` the number of periods. parameter is estimated in addition to the arrival rates.

Returns the constructed arrival process.

```
public static DirichletArrivalProcess getInstanceFromMLE
(PeriodChangeEvent pce, ContactFactory factory, RandomStream stream,
RandomStream streamArr, Class<? extends Distribution> aDistClass, int[] []
arrivals, int numObs, int numPeriods)
```

Similar to `getInstanceFromMLE (PeriodChangeEvent, ContactFactory, RandomStream, RandomVariateGen, int[] [], int, int)`, but also estimates the parameters for A . This method accepts a class object `aDistClass` which is the guessed probability distribution of A . It uses `DistributionFactory` to get an instance of the distribution (with estimated parameters), and constructs the arrival process by using this distribution, and the Dirichlet parameters estimated by `getMLE (int[] [], int, int)`.

Parameters

`pce` the period-change event marking the end of periods.

`factory` the contact factory used to create contacts.

`stream` the random stream to generate arrival times.

`streamArr` the random stream for A .

`aDistClass` the class of the probability distribution of A .

`arrivals` the number of arrivals.

`numObs` the number of days.

`numPeriods` the number of periods. parameter is estimated in addition to the arrival rates.

Returns the constructed arrival process.

NORTADrivenArrivalProcess

Represents an arrival process in which the numbers of arrivals per-period are correlated negative binomial random variables, generated using the NORTA method. To generate the number of arrivals, the process first obtains a vector $\mathbf{X} = (X_1, \dots, X_P)$ from the multivariate normal distribution with mean vector $\mathbf{0}$ and covariance matrix Σ . Assuming that Σ is a correlation matrix, i.e., each element is in $[-1, 1]$ and 1's are on its diagonal, the vector of uniforms $\mathbf{U} = (\Phi(X_1), \dots, \Phi(X_P))$ is obtained, where $\Phi(x)$ is the distribution function of a standard normal variable. For main period p , the marginal probability distribution for A_p is assumed to be negative binomial with parameters γ_p and ρ_p , γ_p being a positive number and $0 < \rho_p < 1$. A_0 and A_{P+1} , the number of arrivals during the preliminary and the wrap-up periods, respectively, are always 0 for this process.

Since the numbers of arrivals per-period are generated directly, this process does not arise as a Poisson arrival process. However, inter-arrival times are generated as if $A_p^* = \text{round}(BA_p)$ was a Poisson variate. As a result, for each main period, the arrival process generates A_p^* uniforms ranging from the beginning to the end of the period, and the uniforms are sorted to get inter-arrival times.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class NORTADrivenArrivalProcess extends PoissonUniformArrivalProcess
```

Constructor

```
public NORTADrivenArrivalProcess (PeriodChangeEvent pce, ContactFactory
                                factory, DoubleMatrix2D sigma, double[]
                                gammas, double[] probs, RandomStream
                                stream)
```

Constructs a new NORTA-driven arrival process with period-change event `pce`, contact factory `factory`, correlation matrix `sigma`, negative binomial parameters (`gammas[p]`, `probs[p]`), and random number stream `stream`.

Parameters

`pce` the period-change event defining the periods.

`factory` the contact factory instantiating contacts.

`sigma` the correlation matrix.

`gammas` the γ parameters for negative binomials.

`probs` the ρ parameters for negative binomials.

`stream` the random number stream for correlated negative binomial vectors and uniform arrival times.

Throws

`NullPointerException` if one argument is null.

`IllegalArgumentException` if the dimensions of the correlation matrix does not correspond to $P \times P$, or the length of `ns` or `probs` do not correspond to number of main periods P .

Methods

```
public DoubleMatrix2D getSigma()
```

Returns the correlation matrix associated with this arrival process.

Returns the associated correlation matrix.

```
public void setSigma (DoubleMatrix2D sigma)
```

Sets the associated correlation matrix to `sigma`.

Parameter

`sigma` the new correlation matrix.

Throws

`NullPointerException` if `sigma` is null.

`IllegalArgumentException` if `sigma` is not a $P \times P$ symmetric and positive-definite matrix.

```
public double getNegBinGamma (int p)
```

Returns the value of γ_p , the negative binomial double-precision parameter associated with main period p .

Parameter

`p` the main period index.

Returns the value of γ_p .

Throws

`ArrayIndexOutOfBoundsException` if `p` is negative or greater than or equal to P .

```
public double getNegBinP (int p)
```

Returns the value of ρ_p , the negative binomial double-precision parameter associated with main period p .

Parameter

`p` the main period index.

Returns the value of ρ_p .

Throws

`ArrayIndexOutOfBoundsException` if `p` is negative or greater than or equal to P .

```
public void setNegBinParams (int p, double gammap, double rhop)
```

Sets the parameters for the negative binomial of period p to γ_p and ρ_p .

Parameters

`p` the index of the main period.

`gammap` the new value of γ_p .

`rhop` the new value of ρ_p .

Throws

`ArrayIndexOutOfBoundsException` if `p` is negative or greater than or equal to P .

`IllegalArgumentException` if the negative binomial parameters are invalid.

CorrelationMatrixCorrector

Implements the algorithm POSDEF that transforms the approximate correlation matrix that may contain negative eigenvalues to the valid positive definite correlation matrix. The algorithm is described in Davenport, J. M. and Iman, R. L. (1982). An iterative algorithm to produce a positive definite correlation matrix from an approximate correlation matrix. Technical report, Sandia National Laboratories, Albuquerque, New Mexico. If the POSDEF algorithm fails to converge this implementation uses diagonal loading algorithm. In this case it returns the original matrix plus scaled identity matrix. The returned matrix is scaled so that the diagonal entries are equal to 1.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class CorrelationMatrixCorrector
```

Constructors

```
public CorrelationMatrixCorrector (int maxit, double epsilon, double[] []  
                                   Rin)
```

Constructs CorrelationMatrixCorrector object from the correlations matrix Rin. Sets maximum number of iterations equal to maxit and the number, which is used to substitute the negative eigenvalues, equal to epsilon. The correlation matrix Rin must be symmetric.

Parameters

maxit Maximum number of iterations

epsilon Number used to replace the negative eigenvalues

Rin Input correlation matrix

```
public CorrelationMatrixCorrector (double[] [] Rin)
```

Constructs CorrelationMatrixCorrector object from the correlations matrix Rin. Sets **maxit** and **epsilon** to default values of 100 and 1e-3 respectively.

Parameter

Rin the input correlation matrix

Methods

```
public double[] [] getRout()
```

Returns the corrected positive definite correlation matrix.

Returns the positive definite correlation matrix

```
public double getEpsilon()
```

Returns the number, which is used to substitute for the negative eigenvalues.

Returns value replacing negative eigenvalues

```
public int getMaxit()
```

Returns the maximum number of iterations in the algorithm.

Returns maximum number of iterations

```
public void setMaxit (int maxit)
```

Sets the maximum number of iterations in the algorithm.

Parameter

`maxit` The maximum number of iterations in the algorithm.

```
public void setEpsilon (double epsilon)
```

Sets the number, which is used to substitute the negative eigenvalues in the POSDEF algorithm

Parameter

`epsilon` The positive number, which is used to substitute the negative eigenvalues in the POSDEF algorithm

```
public void setEpsilonLoading (double epsilonLoading)
```

Sets the number, which is used to scale the identity matrix in the diagonal loading algorithm

Parameter

`epsilonLoading` The positive number, which is used to scale the identity matrix in the diagonal loading algorithm

```
public double[][] calcCorrectedR()
```

Calculate the corrected correlation matrix according to the posdef algorithm proposed by Davenport and Iman.

Returns Corrected correlation matrix in 2D double array.

CorrelationMtxFitting

Fits the parametric model for the correlation matrix using method of least squares. Two models are implemented: (i) general linear model $r_j = ab^j + c$, (ii) single rho Markov model $r_j = b^j$. Method `fitMarkovGeneralLinear()` implements fitting of model (i), method `fitMarkovSingleRho()` implements fitting of model (ii). The optimization of parameter b is performed using exhaustive grid search with step `step` in the range `[-1+delta, 1-delta]`. For model (i) parameters a and c have closed form expressions in terms of the entries of correlation matrix and parameter b .

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class CorrelationMtxFitting
```

Methods

```
public void setDelta (double delta)
```

Sets the limits in which exhaustive grid search for the optimization of parameter b is performed. The search is performed in the interval `[-1+delta, 1-delta]`.

Parameter

`delta` Exhaustive grid search boundary offset

```
public void setStep (double step)
```

Sets the exhaustive grid search grid size for the optimization of parameter b .

Parameter

`step` Exhaustive grid search grid size

```
public double[] fitMarkovGeneralLinear()
```

Fits general linear model $r_j = ab^j + c$. Returns vector of length 3 with parameters a , b and c .

Returns vector of $[a, b$ and $c]$.

```
public double fitMarkovSingleRho()
```

Fits the Markov model with single correlation coefficient of the form $\rho_j = b^j$.

Returns parameter b .

PoissonArrivalProcessWithInversion

Defines a Poisson arrival process with arrival rate $B\lambda(t)$ at time t and generated by inversion. If

$$B\Lambda(t) = \int_0^t B\lambda(s)ds$$

is the cumulative arrival rate of the Poisson process, and $M(t) = N(\Lambda^{-1}(t)/B)$, $\{M(t), t \geq 0\}$ is a standard Poisson process, i.e., homogeneous with arrival rate 1. If $\Lambda^{-1}(t)$ can be computed easily, this class generates arrival times by inversion as follows: generate the arrival times X_0, X_1, \dots for a standard Poisson process and let $T_j = \Lambda^{-1}(X_j)/B$ be the arrival times of the non-homogeneous Poisson process.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class PoissonArrivalProcessWithInversion extends PoissonArrivalProcess
```

Constructors

```
public PoissonArrivalProcessWithInversion (ContactFactory factory,
                                           RandomStream stream,
                                           MathFunction cLambda,
                                           MathFunction invLambda)
```

Constructs a new transformed Poisson arrival process using contact factory `factory` for creating contacts, random stream `stream` for generating uniforms, `cLambda` for the $\Lambda(t)$ function, and `invLambda` for the $\Lambda^{-1}(t)$ function.

Parameters

`factory` the contact factory used to create contacts.

`stream` the random stream used to generate uniforms.

`cLambda` the function defining $\Lambda(t)$.

`invLambda` the function defining $\Lambda^{-1}(t)$.

Throws

`NullPointerException` if any argument is null.

```
public PoissonArrivalProcessWithInversion (Simulator sim, ContactFactory
                                           factory, RandomStream stream,
                                           MathFunction cLambda,
                                           MathFunction invLambda)
```

Equivalent to `PoissonArrivalProcessWithInversion (ContactFactory, RandomStream, MathFunction, MathFunction)`, using the given simulator `sim`.

```
public PoissonArrivalProcessWithInversion (ContactFactory factory,
                                           RandomStream stream,
                                           MathFunction cLambda)
```

Similar to `PoissonArrivalProcessWithInversion (factory, stream, cLambda, f)`, where `f` is a function performing the inversion of `cLambda` using the Brent-Decker root finding algorithm. This can be used when the $\Lambda^{-1}(t)$ function is unavailable, and $\Lambda(t)$ can be computed efficiently. However, the generated inversion function can be slow to compute.

Parameters

`factory` the contact factory used to create contacts.

`stream` the random stream used to generate uniforms.

`cLambda` the function defining $\Lambda(t)$.

Throws

`NullPointerException` if any argument is null.

```
public PoissonArrivalProcessWithInversion (Simulator sim, ContactFactory
                                           factory, RandomStream stream,
                                           MathFunction cLambda)
```

Equivalent to `PoissonArrivalProcessWithInversion (ContactFactory, RandomStream, MathFunction)`, using the given simulator `sim`.

Methods

```
public MathFunction getCumulativeLambdaFunction()
```

Returns the function $\Lambda(t)$ in use.

Returns the $\Lambda(t)$ function.

```
public void setCumulativeLambdaFunction (MathFunction cLambda)
```

Sets the $\Lambda(t)$ function to `cLambda`.

Parameter

`cLambda` the new $\Lambda(t)$ function.

Throws

`NullPointerException` if `cLambda` is null.

```
public MathFunction getInvertedLambdaFunction()
```

Returns the function $\Lambda^{-1}(t)$ in use.

Returns the $\Lambda^{-1}(t)$ function.

```
public void setInvertedLambdaFunction (MathFunction invLambda)
```

Sets the $\Lambda^{-1}(t)$ function to `invLambda`. If `invLambda` is null, the method sets the current $\Lambda^{-1}(t)$ to the default inversion function, which uses the Brent-Decker root finder.

Parameter

`invLambda` the new $\Lambda^{-1}(t)$ function.

PoissonArrivalProcessWithThinning

Defines a Poisson arrival process with arrival rate $B\lambda(t) \leq B\bar{\lambda}$ for time t , and generated using the thinning method. This arrival process generates pseudo-arrivals as a homogeneous Poisson process with rate $B\bar{\lambda}$. A pseudo-arrival at time t is accepted, i.e., becomes an arrival, with probability $\lambda(t)/\bar{\lambda}$, and rejected with probability $1 - \lambda(t)/\bar{\lambda}$.

```
package umontreal.iro.lecuyer.contactcenters.contact;
```

```
public class PoissonArrivalProcessWithThinning extends PoissonArrivalProcess
```

Constructors

```
public PoissonArrivalProcessWithThinning (ContactFactory factory,
                                           RandomStream stream,
                                           RandomStream uStream,
                                           MathFunction lambda, double
                                           lambdaMax, double maxTime)
```

Constructs a new thinned Poisson arrival process using `factory` to generate contacts, `stream` to generate pseudo-arrivals, `uStream` to test for acceptance or rejection, `lambda` for $\lambda(t)$, and `lambdaMax` for $\bar{\lambda}$.

Parameters

`factory` the contact factory used to construct contacts.

`stream` the random stream for pseudo-arrivals.

`uStream` the random stream for tests of acceptance.

`lambda` the function $\lambda(t)$.

`lambdaMax` the value of $\bar{\lambda}$.

`maxTime` the smallest time T for which $\lambda(t) = 0$ for any $t \geq T$.

Throws

`NullPointerException` if any argument is null.

`IllegalArgumentException` if `lambdaMax` is negative, infinite, or NaN, or if `maxTime` is negative.

```
public PoissonArrivalProcessWithThinning (Simulator sim, ContactFactory
                                           factory, RandomStream stream,
                                           RandomStream uStream,
                                           MathFunction lambda, double
                                           lambdaMax, double maxTime)
```

Equivalent to `PoissonArrivalProcessWithThinning (ContactFactory, RandomStream, RandomStream, MathFunction, double, double)`, using the given simulator `sim`.

Methods

`public MathFunction getLambdaFunction()`

Returns the $\lambda(t)$ function.

Returns the $\lambda(t)$ function.

`public void setLambdaFunction (MathFunction lambda)`

Sets the $\lambda(t)$ function to `lambda`.

Parameter

`lambda` the new $\lambda(t)$ function.

Throws

`NullPointerException` if `lambda` is null.

`public double getLambda()`

Returns the value of $\bar{\lambda}$.

`public void setLambda (double lambda)`

Sets the value of $\bar{\lambda}$ to `lambda`.

Throws

`IllegalArgumentException` if `lambdaMax` is negative, infinite, or NaN.

`public RandomStream getRejectionStream()`

Returns the random stream for tests of acceptance.

Returns the random stream for tests of acceptance.

`public void setRejectionStream (RandomStream uStream)`

Sets the random stream for tests of acceptance to `uStream`.

Parameter

`uStream` the new random stream for tests of acceptance.

Throws

`NullPointerException` if `uStream` is null.

`public double getMaximalTime()`

Returns the smallest time T for which $\lambda(t) = 0$ for all $t \geq T$. This corresponds to the maximal time an arrival can occur.

Returns the maximal time.

`public void setMaximalTime (double maxTime)`

Sets the maximal time T to `maxTime`.

Parameter

`maxTime` the new maximal time T .

Throws

`IllegalArgumentException` if `maxTime` is negative.

Package `umontreal.iro.lecuyer.contactcenters.queue`

Manages waiting queues for storing contacts not being served immediately. A waiting queue represents a data structure organizing waiting contacts while supporting abandonment. Three types of data structures are available for waiting queues, each implemented in concrete subclasses of `WaitingQueue`: a list, a sorted set, or a heap.

The `StandardWaitingQueue` uses a `List` for First In First Out (FIFO) or Last In First Out (LIFO) queues. Queued contacts are ordered based on their arrival times, and can be easily enumerated. When the number of priorities is finite and small, priority queues can be implemented efficiently by combining several standard waiting queues.

General priority queues are implemented by `QueueWaitingQueue`, which uses a heap implemented by the `Queue` interface. By default, contacts are ordered using their arrival times and their priorities, but the user may supply its own `Comparator` to change this order. Heaps are very efficient for inserting new contacts or removing the first contact, but queued contacts are not enumerated in any particular order.

The most generic waiting queue, `PriorityWaitingQueue`, uses a `SortedSet` to store contacts. As with the heap-based queue, the user can supply its own `Comparator` to order contacts. Contacts are also enumerated in the order defined by the comparator used. However, the operations on the sorted set are slower than the operations on a list or a heap.

This package also provides support classes to obtain maximal queue times and to compute the integral of the queue size over simulation time.

WaitingQueue

Represents a waiting queue where contacts are added if they cannot be served immediately. The queue contains `DequeueEvent` objects being scheduled to happen at the time of automatic removal, e.g., abandonment, disconnection, etc. These dequeue events, which encapsulate contacts, are used to support abandonment as well as other types of exits of queue. When a contact is added at the end of the queue using the `add (Contact)` method, its dequeue event is constructed, and scheduled if a maximal queue time is available. If the dequeue event occurs, the associated queued contact is removed from the queue. Queued contacts can also be removed manually using the `removeFirst (int)` or `removeLast (int)` methods (this cancels the appropriate dequeue event), or visited by an iterator returned by `iterator (int)`. An iterator is useful to enumerate queued contacts, and to remove arbitrary ones.

All registered *waiting-queue listeners* are notified about added and removed contacts. The reason of the removal is available for listeners through an integer called the *dequeue type*, encapsulated in the dequeue event. For example, this permits statistical collectors to distinguish abandonment from disconnection.

This abstract class does not implement a data structure for storing queued contacts. The subclasses `StandardWaitingQueue`, `QueueWaitingQueue`, and `PriorityWaitingQueue` implement such data structures.

Note: the `WaitingQueueListener` implementations are notified in the order of the list returned by `getWaitingQueueListeners()`, and a waiting-queue listener modifying the list of listeners by using `addWaitingQueueListener (WaitingQueueListener)` or `removeWaitingQueueListener (WaitingQueueListener)` could result in unpredictable behavior.

```
package umontreal.iro.lecuyer.contactcenters.queue;

public abstract class WaitingQueue extends AbstractQueue<DequeueEvent>
    implements Initializable, Named
```

Field

```
protected int dqTypeRet
```

Contains the dequeue type generated by `getMaximalQueueTime (DequeueEvent)`.

Constructor

```
public WaitingQueue()
```

Constructs a new waiting queue.

Methods

```
public void init()
```

Initializes this waiting queue for a new simulation replication. This removes all the contacts from the queue without notification of individual contacts to the listeners.

```
public void clear (int dqType)
```

Removes all the contacts contained into this waiting queue with dequeue type `dqType`. In contrast with `init()`, any removed contact is notified to the registered listeners.

Parameter

`dqType` the dequeue type of the removed contacts.

```
public int size()
```

Returns the number of contacts in this waiting queue.

Returns the number of contacts in the waiting queue.

```
public int size (int k)
```

Returns the number of contacts of type `k` in this waiting queue.

Parameter

`k` the tested contact type.

Returns the number of contacts of type `k` in the queue.

```
public DequeueEvent add (Contact contact)
```

Adds the contact `contact` to the waiting queue and returns a reference to the constructed dequeue event. The maximal queue time is obtained using `getMaximalQueueTime (DequeueEvent)`, and an event is scheduled with its corresponding dequeue type. In case of a zero or negative maximal queue time, the contact is enqueued then immediately dequeued. Otherwise, the contact is enqueued and a dequeue event is scheduled if the maximal queue time is not `Double.POSITIVE_INFINITY` or `Double.NaN`. The returned event can be used to get information about the queued contact and to manually remove it from the queue. If it is directly cancelled, the contact will not leave the queue automatically.

Parameter

`contact` the contact to be added.

Returns a reference to the dequeue event.

```
public DequeueEvent add (Contact contact, double enqueueTime, double
                        maxQueueTime, int dqType)
```

This is the same as `add (Contact)`, except that the enqueue time, maximal queue time and dequeue type if the queue time is reached, are specified explicitly.

Parameters

`contact` the contact being queued.

`enqueueTime` the time at which the contact joined the queue.

`maxQueueTime` the maximal queue time.

`dqType` the dequeue type if the maximal queue time is reached.

Returns the dequeue event representing the queued contact.

```
public DequeueEvent addFromOldEvent (DequeueEvent oldDequeueEvent)
```

Adds a contact into the queue by using the information stored in an old dequeue event `oldDequeueEvent`. This method extracts the queued contact, the scheduled maximal queue time, and the scheduled dequeue type from the given event, and uses that information to call `add (Contact, double, double, int)`.

Parameter

`oldDequeueEvent` the old dequeue event.

Returns the new dequeue event representing the queued contact.

```
protected double getMaximalQueueTime (DequeueEvent ev)
```

Generates and returns the maximal queue time for the queued contact represented by `ev`. The method can store a dequeue type in the protected field `dqTypeRet` if the default value of 1 is not appropriate.

By default, a `MinValueGenerator` is used. For each dequeue type q with an associated value generator, a maximal queue time V_q is generated. The scheduled queue time is $V_{q^*} = \min_q \{V_q\}$, and the dequeue type is q^* .

Parameter

`ev` the dequeue event representing the queued contact.

Returns the maximal queue time.

```
public boolean remove (DequeueEvent dqEvent, int dqType)
```

Removes the contact identified by the dequeue event `dqEvent`, setting its effective dequeue type to `dqType`. Returns `true` if the removal was successful, `false` otherwise.

Parameters

`dqEvent` the dequeue event.

`dqType` the effective dequeue type.

Returns the success indicator of the operation.

```
public boolean remove (Contact contact, int dqType)
```

Removes the contact `contact` from the waiting queue. with dequeue type `dqType`. Returns `true` if the contact was removed, `false` otherwise. If a dequeue event was scheduled when the contact was added, this event is cancelled. This method has to linearly search for the contact being removed using `getDequeueEvent (Contact)`, which is less efficient than when a dequeue event is given.

Parameters

`contact` the contact being removed from the queue.

`dqType` the effective dequeue type of the contact.

Returns `true` if the contact was removed, `false` otherwise.

```
public DequeueEvent getDequeueEvent (Contact contact)
```

Returns the dequeue event for the contact `contact`. If the contact is not in queue, this returns `null`. Since this method has to perform a linear search, it is more efficient to keep the dequeue events returned by `add (Contact)` when they are needed.

Parameter

`contact` the queried contact.

Returns the dequeue event for the contact, or `null` if the contact was not found.

```
public DequeueEvent getFirst()
```

Returns the dequeue event representing the first contact in the queue, or throws a `NoSuchElementException` if the queue is empty.

Returns the dequeue event for the first contact in the queue.

Throws

`NoSuchElementException` if the queue is empty.

```
public DequeueEvent getLast()
```

Returns the dequeue event representing the last contact in the queue, or throws a `NoSuchElementException` if the queue is empty.

Returns the dequeue event for the last contact in the queue.

Throws

`NoSuchElementException` if the queue is empty.

```
public DequeueEvent removeFirst (int dqType)
```

Removes the first contact in the waiting queue and returns the corresponding dequeue event. The event is assigned the effective dequeue type `dqType`. If the queue is empty, a `NoSuchElementException` is thrown. The `getFirst()` method is used to get the dequeue event.

Parameter

`dqType` the effective dequeue type.

Returns the dequeue event corresponding to the removed contact.

Throws

`NoSuchElementException` if the queue is empty.

```
public DequeueEvent removeLast (int dqType)
```

Removes the last contact in the waiting queue and returns the corresponding dequeue event. The event is assigned the effective dequeue type `dqType`. If the queue is empty, a `NoSuchElementException` is thrown. The `getLast()` method is used to get the dequeue event.

Parameter

`dqType` the effective dequeue type.

Returns the dequeue event corresponding to the removed contact.

Throws

`NoSuchElementException` if the queue is empty.

```
public Iterator<DequeueEvent> iterator (int dqType)
```

Returns an iterator allowing the dequeue events representing contacts in queue to be enumerated. The order of the elements depends on the type of waiting queue and the order of insertion. The objects returned by the iterator's `next()` method are instances of the `DequeueEvent` class. The optional `remove()` method is implemented and removes contacts with dequeue type `dqType`. If `remove()` is never called on the returned iterator, `dqType` is not used.

Parameter

`dqType` the dequeue type of any removed contact.

Returns an iterator enumerating the contacts in queue.

```
public Iterator<DequeueEvent> iterator()
```

This is similar to `iterator (int)`, except it uses the default dequeue type returned by `getDefaultDequeueType()`.

Returns the constructed iterator.

```
public int getDefaultDequeueType()
```

Returns the default dequeue type used by this object when the user does not specify a dequeue type explicitly. The initial default dequeue type is 0.

Returns the default dequeue type.

```
public void setDefaultDequeueType (int dqTypeDefault)
```

Sets the default dequeue type to `dqTypeDefault`.

Parameter

`dqTypeDefault` the new default dequeue type.

```
public void addWaitingQueueListener (WaitingQueueListener listener)
```

Adds the new waiting-queue listener `listener` to this object. If the listener is already added, nothing happens; it is not added a second time.

Parameter

`listener` the listener being added.

Throws

`NullPointerException` if `listener` is `null`.

```
public void removeWaitingQueueListener (WaitingQueueListener listener)
```

Removes the waiting-queue listener `listener` from this object. If the listener is not registered, nothing happens.

Parameter

`listener` the waiting-queue listener being removed.

```
public void clearWaitingQueueListeners()
```

Removes all waiting-queue listeners registered with this waiting queue.

```
public List<WaitingQueueListener> getWaitingQueueListeners  
( )
```

Returns an unmodifiable list containing all the waiting-queue listeners registered with this waiting queue.

Returns the list of all registered waiting-queue listeners.

```
protected void notifyInit()
```

Notifies every registered listener that this waiting queue was initialized.

```
protected void notifyEnqueued (DequeueEvent ev)
```

Notifies every registered listener that a contact was enqueued, this event being represented by `ev`.

Parameter

`ev` the dequeue event representing the queued contact.

```
protected void notifyDequeued (DequeueEvent ev)
```

Notifies every registered listener that a contact left this queue, this event being represented by `ev`.

Parameter

`ev` the event representing the contact having left the queue.

```
public WaitingQueueState save()
```

Constructs a new `WaitingQueueState` object holding the current state of this waiting queue, i.e., every queued contact.

Returns the state of this waiting queue.

```
public void restore (WaitingQueueState state)
```

Restores the state of the waiting queue by using the `restore` method of `state`.

Parameter

`state` the saved state of the waiting queue.

```
public int getId()
```

Returns the identifier associated with this queue. This identifier, which defaults to `-1`, can be used as an index in routers.

Returns the identifier associated with this queue.

```
public void setId (int id)
```

Sets the identifier of this queue to `id`. Once this identifier is set to a positive or 0 value, it cannot be changed anymore. This method is automatically called by the router when a waiting queue is connected. If one tries to attach the same queue to different routers, the queue must have the same index for each of them. For this reason, if one tries to change the identifier, an `IllegalStateException` is thrown.

Parameter

`id` the new identifier associated with the queue.

Throws

`IllegalStateException` if the identifier was already set.

```
public ValueGenerator getMaximalQueueTimeGenerator (int dqType)
```

Returns the maximal queue time generator associated with dequeue type `dqType` for this waiting queue. Returns `null` if no value generator is associated with the given `dqType`.

Parameter

`dqType` the queried dequeue type.

Returns the maximal queue time generator of this object.

```
public void setMaximalQueueTimeGenerator (int dqType, ValueGenerator dqgen)
```

Changes the maximal queue time generator associated with dequeue type `dqType` for this waiting queue to `dqgen`.

Parameters

dqType the affected dequeue type.

dqgen the new maximal queue time generator.

Throws

IllegalArgumentException if the given dequeue type is negative.

NullPointerException if **dqgen** is null.

```
protected abstract Iterator<DequeueEvent> elementsIterator  
( )
```

Returns an iterator capable of traversing, in the correct order, the elements in the waiting queue's internal data structure. This is different from the **iterator (int)** method because this iterator returns the contacts marked for dequeue as well as the contacts still enqueued. If the returned iterator does not implement **remove()**, **remove (Contact, int)** and **get-DequeueEvent (Contact)** will not work properly.

Returns an iterator for the waiting queue elements.

```
protected abstract void elementsClear()
```

Clears all elements in the data structure representing the queued contacts.

```
protected abstract void elementsAdd (DequeueEvent dqEvent)
```

Adds the new dequeued event **dqEvent** to the internal data structure representing the waiting queue.

Parameter

dqEvent the dequeue event being added.

```
protected abstract boolean elementsIsEmpty()
```

Determines if the internal waiting queue data structure is empty.

Returns **true** if the data structure is empty, **false** otherwise.

```
protected abstract DequeueEvent elementsGetFirst()
```

Returns the first element of the waiting queue's internal data structure, or throws a **NoSuchElementException** if no such element exists.

Returns the first element of the data structure.

Throws

NoSuchElementException if the queue's data structure is empty.

```
protected abstract DequeueEvent elementsGetLast()
```

Returns the last element of the waiting queue's internal data structure, or throws a **NoSuchElementException** if no such element exists.

Returns the last element of the data structure.

Throws

`NoSuchElementException` if the queue's data structure is empty.

`protected abstract DequeueEvent elementsRemoveFirst()`

Removes and returns the first element in the waiting queue's internal data structure. Throws a `NoSuchElementException` if no such element exists.

Returns the removed element.

Throws

`NoSuchElementException` if the queue's data structure is empty.

`protected abstract DequeueEvent elementsRemoveLast()`

Removes and returns the last element in the waiting queue's internal data structure. Throws a `NoSuchElementException` if no such element exists.

Returns the removed element.

Throws

`NoSuchElementException` if the queue's data structure is empty.

`public Map<Object, Object> getAttributes()`

Returns the map containing the attributes for this waiting queue. Attributes can be used to add user-defined information to waiting queue objects at runtime, without creating a subclass. However, for maximal efficiency, it is recommended to create a subclass of `WaitingQueue` instead of using attributes.

Returns the map containing the attributes for this object.

DequeueEvent

Represents an event happening when a contact leaves a waiting queue without being explicitly removed. This event also holds the necessary information about a contact in queue and is added to the waiting queue's data structure. When it becomes obsolete, it can be used to keep track of the queueing step of the concerned contact. For this reason, the event implements the `ContactStepInfo` interface.

Note that the natural ordering of dequeue events corresponds to ascending order of automatic removal from queue, not the order of insertion. This is adapted for insertion of dequeue events in event lists, not for priority queues. The class `DequeueEventComparator` must be used to impose the order of insertion, for priority queues. This comparator is used when calling the default constructor of `PriorityWaitingQueue` and `QueueWaitingQueue`.

```
package umontreal.iro.lecuyer.contactcenters.queue;

public final class DequeueEvent extends Event
    implements ContactStepInfo, Cloneable
```

Constructor

```
protected DequeueEvent (WaitingQueue queue, Contact contact, double
                        enqueueTime)
```

Constructs a new dequeue event with contact `contact` entering waiting queue `queue` at simulation time `enqueueTime`.

This constructor is rarely used directly; the recommended way to create dequeue events is to use `WaitingQueue.add (Contact)`.

Parameters

`queue` the associated waiting queue.

`contact` the contact being queued.

`enqueueTime` the time at which the contact enters the queue.

Methods

```
public Contact getContact()
```

Returns a reference to the queued contact.

Returns a reference to the queued contact.

```
public WaitingQueue getWaitingQueue()
```

Returns a reference to the waiting queue.

Returns a reference to the waiting queue.

```
public double getEnqueueTime()
```

Returns the simulation time at which the contact was enqueued.

Returns the contact's enqueue time.

```
public double getScheduledQueueTime()
```

Returns the scheduled queue time for this contact. This corresponds to the maximal time the contact can spend in queue before being automatically removed, if this event occurs.

Returns the contact's scheduled queue time.

```
public int getScheduledDequeueType()
```

Returns the scheduled dequeue type of the contact if this event occurs. This scheduled dequeue type can be overridden when a contact is manually removed.

Returns the contact's scheduled dequeue type.

```
public void setScheduledDequeueType (int dqType)
```

Changes the dequeue type of the contact to `dqType` when the event occurs. If this is called after the contact was dequeued, an `IllegalStateException` is thrown.

Parameter

`dqType` the new type of removal.

Throws

`IllegalStateException` if the contact was dequeued.

```
public double getEffectiveQueueTime()
```

Returns the simulation time the contact has effectively spent in queue.

Returns the effective queue time.

Throws

`IllegalStateException` if the contact is still in queue.

```
public int getEffectiveDequeueType()
```

Returns the effective dequeue type of the contact having waited in this queue. Throws an `IllegalStateException` if the contact is still in queue.

Returns the effective dequeue type.

Throws

`IllegalStateException` if the contact is still in queue.

```
public boolean remove (int dqType1)
```

Removes this dequeue event from its associated waiting queue, with dequeue type `dqType`. Returns `true` if and only if the removal was successful. This method calls `getWaitingQueue().remove (this, dqType)`, and returns the result.

Parameter

`dqType1` the dequeue type.

Returns the success indicator of the operation.

`public boolean dequeued()`

Indicates that the contact has left the queue and that this event is obsolete. If an obsolete event is scheduled, an `IllegalStateException` is thrown at the time it happens. The event can be used as a data structure to keep a trace of the queueing process of the contact.

Returns the dequeue indicator.

`public boolean isObsolete()`

Determines if this event is obsolete. When calling `WaitingQueue.init()`, some dequeue events might still be in the simulator's event list. One must use this method in `actions()` to test if this event is obsolete. If that returns `true`, one should return immediately.

Returns `true` for an obsolete event, `false` otherwise.

`public int compareTo (DequeueEvent ev)`

Compares this dequeue event with the other event `ev`. The method extracts the `Contact` object from this event and from the `ev` argument. The `Contact.compareTo (Contact)` method is then used to compare objects. A contact that cannot be extracted is assigned the null value and precedes any non-null contacts.

Parameter

`ev` the other event being compared.

Returns the result of the comparison.

`public DequeueEvent clone()`

Returns a copy of this event. This method clones every field of the event, except the waiting queue which is not cloneable.

`public DequeueEvent clone (Contact clonedContact)`

Similar to `clone()`, but initializes the contact of the cloned event with `clonedContact` instead of a clone of the contact returned by `getContact()`. This method can be useful when cloning a contact `c` for which `c.getSteps()` returns a non-empty list containing dequeue events. In that case, the contact associated with the events included in `c.getSteps()` must correspond to `c` rather than clones of `c`.

DequeueEventComparator

Default comparator used to sort dequeue events in a priority queue. The default order for `DequeueEvent` is given by the `Comparable.compareTo (T)` method, which sorts events according to time of occurrence, i.e., dequeue time. This is adapted for inserting dequeue events in the event list of the simulator, not in a waiting queue. This comparator can be used when the waiting queue needs a comparator to establish the order of the elements. This comparator is not needed for waiting queues using a list, i.e., `StandardWaitingQueue`.

```
package umontreal.iro.lecuyer.contactcenters.queue;  
  
public class DequeueEventComparator implements Comparator<DequeueEvent>
```

Method

```
public int compare (DequeueEvent e1, DequeueEvent e2)
```

Compares dequeue event `e1` with the other event `e2`. The method extracts the `Contact` objects from the events. The `Contact.compareTo (Contact)` method is then used to compare objects. A contact that cannot be extracted is assigned the `null` value and precedes any non-null contacts.

Parameters

`e1` the first event.

`e2` the second event.

Returns the result of the comparison.

WaitingQueueSet

Represents a group of waiting queues for which it is possible to get the total size. This can be used when the total number of contacts in a subset of the contact center's waiting queues is needed for statistical collecting or for capacity limitation.

```
package umontreal.iro.lecuyer.contactcenters.queue;  
  
public class WaitingQueueSet extends AbstractSet<WaitingQueue>  
    implements Initializable, Named, Cloneable
```

Methods

```
public int queueSize()
```

Returns the total size of the queues currently in this group of waiting queues.

Returns the size of all contained queues.

```
public boolean add (WaitingQueue queue)
```

Adds the waiting queue `queue` to this set of waiting queues.

Parameter

`queue` the waiting queue being added.

Throws

`NullPointerException` if `queue` is null.

```
public boolean remove (Object queue)
```

Removes the waiting queue `queue` from this set of waiting queues.

Parameter

`queue` the waiting queue being removed.

Throws

`NullPointerException` if `queue` is null.

```
public void clear()
```

Removes all the waiting queues contained in this set of waiting queues.

```
public void init()
```

Initializes all the waiting queues contained in this set.

```
public void initStat()
```

Initializes the statistical collector for the size of the queues in this set. If statistical collecting is turned OFF, this throws an `IllegalStateException`.

Throws

`IllegalStateException` if statistical collecting is turned OFF.

```
public boolean isStatCollecting()
```

Determines if this set of waiting queues is collecting statistics about the total size of the queues. If this returns `true`, statistical collecting is turned ON. Otherwise (the default), it is turned OFF.

Returns the state of statistical collecting.

```
public void setStatCollecting (boolean b)
```

Sets the state of statistical collecting to `b`. If `b` is `true`, statistical collecting is turned ON. The statistical collectors are created or reinitialized. If `b` is `false`, statistical collecting is turned OFF.

Parameter

`b` the new state of statistical collecting.

```
public void setStatCollecting (Simulator sim)
```

Enables statistical collecting, but associates the given simulator to the internal accumulate.

Parameter

`sim` the simulator associated to the internal accumulate.

```
public Accumulate getStatQueueSize()
```

Returns the statistical collector for the size of the queues in the set. This returns a non-null value only if statistical collecting was turned ON since this object was constructed.

Returns the queue size statistical collector.

```
public WaitingQueueSet clone()
```

Constructs and returns a copy of this set of waiting queues. This method clones the internal set of waiting queues as well as the statistical collectors if they exist. This does not clone the waiting queues themselves.

Returns a clone of this object.

WaitingQueueListener

Represents a waiting-queue listener which can be notified about events concerning waiting queues. When an implementation is registered to a waiting queue, it is notified when contacts are enqueued and dequeued, or when the queue is initialized.

```
package umontreal.iro.lecuyer.contactcenters.queue;
```

```
public interface WaitingQueueListener
```

Methods

```
public void enqueued (DequeueEvent ev)
```

This method is called after a contact was added to a queue. The event `ev` can be used to access the available information about the queued contact. When this is called, it should be possible to use the waiting-queue iterator to find the contact in the queue. However, if the contact is immediately dequeued, it can be absent from the queue.

Parameter

`ev` the dequeue event associated with the queued contact.

```
public void dequeued (DequeueEvent ev)
```

This method is called when a contact is removed from a waiting queue, `ev` representing the corresponding dequeue event.

Parameter

`ev` the obsolete dequeue event.

```
public void init (WaitingQueue queue)
```

This method is called after the `WaitingQueue.init()` method is called for the waiting queue `queue`.

Parameter

`queue` the queue being initialized.

StandardWaitingQueue

Extends the `WaitingQueue` class for a standard waiting queue, without priority. The queue uses a `List` to store the dequeue events ordered by insertion times. By default, a doubly-linked list is used, which implements insertion and removal of the first and last elements in constant time.

```
package umontreal.iro.lecuyer.contactcenters.queue;  
  
public final class StandardWaitingQueue extends WaitingQueue
```

Constructors

```
public StandardWaitingQueue()
```

Constructs a new waiting queue using a `LinkedList` to store the elements.

```
public StandardWaitingQueue (List<DequeueEvent> list)
```

Constructs a new waiting queue using the given `List` implementation to manage the elements. This list must contain only `DequeueEvent` objects and it will be cleared before being used.

Parameter

`list` the list containing the queued contacts.

PriorityWaitingQueue

Extends the `WaitingQueue` class for a priority waiting queue. The queue uses a `SortedSet` to store the dequeue events, and the user can supply a comparator indicating how to order pairs of elements. By default, the sorted set is implemented using a red black tree [5] which is a binary tree with automatic balancing for more stable search speed. This class should be used only when there are many priorities in the system, and queued contacts needs to be enumerated in a consistent order. If there are only a few degrees of priorities, it is more efficient to use one standard waiting queue per priority. If contacts do not have to be enumerated, using a heap is more efficient.

```
package umontreal.iro.lecuyer.contactcenters.queue;

public final class PriorityWaitingQueue extends WaitingQueue
```

Constructors

```
public PriorityWaitingQueue()
```

Constructs a new waiting queue using a `TreeSet` to store the elements. Dequeue events are compared based on their associated contacts, using `DequeueEventComparator`.

```
public PriorityWaitingQueue (Comparator<? super DequeueEvent> comparator)
```

Constructs a new waiting queue using a `TreeSet` to store the elements, and the given `comparator` to determine how to order pairs of elements. The supplied comparator must be able to compare `DequeueEvent` objects.

Parameter

`comparator` the comparator used to sort the elements.

```
public PriorityWaitingQueue (SortedSet<DequeueEvent> set)
```

Constructs a new waiting queue using the given `SortedSet` implementation to manage the elements. At any given time, this sorted set contains only `DequeueEvent` objects. If no comparator is given, dequeue events are compared based on their associated contacts, using `Contact.compareTo (Contact)`. The given `set` will be cleared before it is used.

Parameter

`set` a sorted set object that will contain the dequeue events.

Method

```
public Comparator<? super DequeueEvent> comparator()
```

Returns the comparator used to compare the dequeue events, or `null` if no comparator was given. This method calls `SortedSet.comparator()` and returns the result.

Returns the associated comparator or `null`.

QueueWaitingQueue

Represents a waiting queue using a Java `Queue` implementation as a data structure. For example, `PriorityQueue` can be used as a queue to have a heap. This can be more efficient than using `PriorityWaitingQueue`, which is backed by a `SortedSet`, but `WaitingQueue.getLast()` and `WaitingQueue.removeLast (int)` are not supported, because `Queue` does not provide any method for getting or removing the last element. Moreover, the iterator does not enumerate queued contacts in a particular order.

```
package umontreal.iro.lecuyer.contactcenters.queue;  
  
public class QueueWaitingQueue extends WaitingQueue
```

Constructors

```
public QueueWaitingQueue()
```

Constructs a waiting queue using a priority heap with `DequeueEventComparator` for dequeue event.

```
public QueueWaitingQueue (Comparator<? super DequeueEvent> comparator)
```

Constructs a new waiting queue using a priority heap with the comparator `comparator`.

Parameter

`comparator` the comparator used to compare events.

```
public QueueWaitingQueue (Queue<DequeueEvent> queue)
```

Constructs a new waiting queue using the queue `queue` as a data structure. The given waiting queue can only contain dequeue events, and is cleared before usage.

Parameter

`queue` the queue being used.

Throws

`NullPointerException` if `queue` is null.

QueueSizeStat

Computes statistics for a specific waiting queue. Using accumulates, this class can compute the integral of the queue size from the last call to `init()` to the current simulation time. Optionally, it can also compute the integral of the number of contacts of each type k in queue.

```
package umontreal.iro.lecuyer.contactcenters.queue;
```

```
public class QueueSizeStat implements Cloneable
```

Constructors

```
public QueueSizeStat (WaitingQueue queue)
```

Constructs a new queue size statistical probe for the waiting queue `queue` and only computing aggregate queue size. This is equivalent to `QueueSizeStat (queue, 0)`.

Parameter

`queue` the observed waiting queue.

```
public QueueSizeStat (Simulator sim, WaitingQueue queue)
```

Equivalent to `QueueSizeStat (WaitingQueue)`, using the given simulator `sim` to construct accumulates.

```
public QueueSizeStat (WaitingQueue queue, int numTypes)
```

Constructs a new queue size statistical probe for the waiting queue `queue` supporting `numTypes` contact types.

Parameters

`queue` the observed waiting queue.

`numTypes` the supported number of contact types.

Throws

`IllegalArgumentException` if the number of contact types is smaller than 0.

```
public QueueSizeStat (Simulator sim, WaitingQueue queue, int numTypes)
```

Equivalent to `QueueSizeStat (WaitingQueue, int)`, using the simulator `sim` to construct accumulates.

Methods

```
public void setSimulator (Simulator sim)
```

Sets the simulator attached to internal accumulates to `sim`.

Parameter

`sim` the new simulator.

Throws

`NullPointerException` if `sim` is `null`.

```
public final WaitingQueue getWaitingQueue()
```

Returns the waiting queue currently associated with this object.

Returns the currently associated waiting queue.

```
public final void setWaitingQueue (WaitingQueue queue)
```

Sets the associated waiting queue to `queue`. If the given queue is `null`, the statistical collector is disabled until a non-`null` waiting queue is given. This can be used during a replication if the integrals must be computed during some periods only.

Parameter

`queue` the new associated waiting queue.

```
public Accumulate getStatQueueSize()
```

Returns the statistical collector for the queue size over the simulation time.

Returns the queue size statistical collector.

```
public int getNumContactTypes()
```

Returns the number of contact types supported by this object.

Returns the number of supported contact types.

```
public Accumulate getStatQueueSize (int type)
```

Returns the statistical collector for the number of contacts of type `type` in the queue.

Parameter

`type` the target contact type.

Returns the size collector for the target type.

Throws

`ArrayIndexOutOfBoundsException` if `type` is negative or greater than or equal to the number of supported contact types.

```
public QueueSizeStat clone()
```

Constructs and returns a clone of this queue-size collector. This method clones the internal statistical collectors, but the clone has no associated waiting queue. This can be used to save the state of the statistical collector for future restoration.

Returns a clone of this object.

QueueSizeStatMeasureMatrix

Queue size statistical collector implementing `MeasureMatrix`. This class extends `QueueSizeStat` and implements the `MeasureMatrix` interface and defines measures for queue sizes. If the object supports $K > 1$ contact types, the measure $0 \leq k < K$ corresponds to the integral of the number of contacts of type k over the simulation time. The measure K corresponds to the integral of the queue size over the simulation time. If $K = 1$, only the integral of the queue size is computed and stored in measure 0. Since this measure matrix supports only one period, it must be combined with `IntegralMeasureMatrix` for the integral of the queue size to be obtained for each period.

```
package umontreal.iro.lecuyer.contactcenters.queue;

public class QueueSizeStatMeasureMatrix extends QueueSizeStat
    implements MeasureMatrix
```

Constructors

```
public QueueSizeStatMeasureMatrix (WaitingQueue queue)
```

Constructs a new queue size statistical probe for the waiting queue `queue` and only computing aggregate queue size. This is equivalent to `QueueSizeStat (queue, 0)`.

Parameter

`queue` the observed waiting queue.

```
public QueueSizeStatMeasureMatrix (Simulator sim, WaitingQueue queue)
```

Equivalent to `QueueSizeStatMeasureMatrix (WaitingQueue)`, using the given simulator `sim` to create internal probes.

```
public QueueSizeStatMeasureMatrix (WaitingQueue queue, int numTypes)
```

Constructs a new queue size statistical probe for the waiting queue `queue` supporting `numTypes` contact types.

Parameters

`queue` the observed waiting queue.

`numTypes` the supported number of contact types.

Throws

`IllegalArgumentException` if the number of contact types is smaller than 0.

```
public QueueSizeStatMeasureMatrix (Simulator sim, WaitingQueue queue, int
    numTypes)
```

Equivalent to `QueueSizeStatMeasureMatrix (WaitingQueue, int)`, using the given simulator `sim` to create internal probes.

Methods

```
public static MeasureSet getQueueSizeIntegralMeasureSet  
(MeasureMatrix[] qscal)
```

Returns a measure set regrouping the queue size integrals for several waiting queues. Row **r** of the resulting matrix corresponds to the queue size integral stored in `qscal[r]`, and the last row contains the total queue size.

Parameter

`qscal` the queue size matrices.

Returns the queue size integral measure set.

```
public static MeasureSet getQueueSizeIntegralMeasureSet  
(MeasureMatrix[] qscal, int numTypes)
```

Returns a measure set regrouping the integrals of the number of contacts of each type in a set of waiting queues. The row `numTypes*q + k` contains the integral of the number of contact of type **k** stored in `qscal[q]` over the simulation time. If the measure set is computing the sum row (the default), row `numTypes*qscal.length + k` corresponds to the integral of the total number of queued contacts of type **k**, over the simulation time.

Parameters

`qscal` the queue size integral matrices.

`numTypes` the number of contact types.

Returns the queue size integral measure set.

ContactPatienceTimeGenerator

Value generator for the patience time of contacts. This implementation simply calls the `Contact.getDefaultPatienceTime()` method to get the patience times. For each new waiting queue, such a value generator is created and used by default.

```
package umontreal.iro.lecuyer.contactcenters.queue;
```

```
public class ContactPatienceTimeGenerator implements ValueGenerator
```

WaitingQueueState

Represents the state of a waiting queue. For now, this state is represented by an array of dequeue events representing queued contacts.

```
package umontreal.iro.lecuyer.contactcenters.queue;  
  
public class WaitingQueueState
```

Constructor

```
protected WaitingQueueState (WaitingQueue queue)
```

Constructs a new state object by saving the state of the waiting queue `queue`.

Parameter

`queue` the queue to be saved.

Method

```
public DequeueEvent[] getQueuedContacts()
```

Returns the array containing the queued contacts in the queue at the time the state was saved.

Returns the state of the queued contacts.

EnqueueEvent

Represents a simulation event that will put a queued contact back in its original waiting queue. This is used for state restoration of a waiting queue.

```
package umontreal.iro.lecuyer.contactcenters.queue;  
  
public class EnqueueEvent extends Event
```

Constructors

```
public EnqueueEvent (DequeueEvent oldDequeueEvent)
```

Constructs a new enqueue event from an old dequeue event using the target queue returned by `DequeueEvent.getWaitingQueue()`.

Parameter

`oldDequeueEvent` the old dequeue event to be used.

```
public EnqueueEvent (WaitingQueue targetQueue, DequeueEvent  
                    oldDequeueEvent)
```

Constructs a new enqueue event from an old dequeue event that will put a queued contact into the target waiting queue `targetQueue`.

Parameters

`targetQueue` the target waiting queue.

`oldDequeueEvent` the old dequeue event to be used.

```
public EnqueueEvent (WaitingQueue targetQueue, Contact contact, double  
                    queueTime, int dqType)
```

Constructs a new enqueue event that will put a contact `contact` into the target waiting queue `targetQueue`. The maximal queue time of the contact will be `queueTime` while its dequeue type is `dqType`.

Parameters

`targetQueue` the target waiting queue.

`contact` the contact being queued.

`queueTime` the maximal queue time.

`dqType` the dequeue type.

Throws

`NullPointerException` if `contact` or `targetQueue` are null.

`IllegalArgumentException` if `queueTime` is negative.

Methods

```
public WaitingQueue getTargetWaitingQueue()
```

Returns the waiting queue in which the previously queued contact will be added by this event.

Returns the target waiting queue.

```
public Contact getContact()
```

Returns the contact to be queued when the event occurs.

Returns the contact being queued.

```
public double getScheduledQueueTime()
```

Returns the scheduled maximal queue time assigned to the contact when it is queued.

Returns the scheduled maximal queue time.

```
public int getScheduledDequeueType()
```

Returns the scheduled dequeue type assigned to the contact when it is queued.

Returns the scheduled dequeue type.

```
public DequeueEvent getNewDequeueEvent()
```

Returns the dequeue event representing the contact put back in the waiting queue. This returns a non-null value only after the execution of the `actions()` method.

Returns the new dequeue event.

Package `umontreal.iro.lecuyer.contactcenters.queuemodel`

Implements different formulae used to compute the delay probability and the service level, assuming some queueing model for the arrival and service rates. Examples of models are the Erlang A , the Erlang B and the Erlang C models. The service level is defined as the probability $Pr\{W \leq \text{awt}\}$, where W is the waiting time and `awt` is the acceptable waiting time.

ErlangC

The Erlang C formula is used to compute the delay probability $Pr\{W > 0\}$ and also the service level, defined as $Pr\{W \leq awt\}$, where W is the waiting time and awt the acceptable waiting time. This formula assumes a $M/M/c$ queueing model such that the arrival and service rates are exponential. The $M/M/c$ assumes one call type and one agent group. This queueing model can be analyzed as a stochastic process $X(t) \in \{0, \dots, c + q\}$ representing the number of calls in the system at time t and where $c + q$ is maximum number of calls in the system. The number of servers is c and the capacity of the queue is q . Calls are blocked when the system is at full, $X(t) = c + q$. It assumes no abandonment due to the impatience of the client.

The rates at each state are :

$$\begin{aligned} \lambda_k &= \lambda, & k = 1, 2, \dots, c + q - 1 \\ \mu_k &= \begin{cases} k\mu, & k = 1, 2, \dots, c - 1 \\ c\mu, & k = c, c + 1, \dots, c + q. \end{cases} \end{aligned}$$

```
package umontreal.iro.lecuyer.contactcenters.queuemodel;
```

```
public class ErlangC
```

Constructors

```
public ErlangC (double arrivalRate, double serviceRate, int capacity)
```

Creates a new instance of ErlangC. Set the capacity to `Integer.MAX_VALUE` for infinite queue capacity.

Parameters

`arrivalRate` the arrival rate.

`serviceRate` the service rate.

`capacity` the capacity of the queue.

```
public ErlangC (double arrivalRate, double serviceRate)
```

Creates a new instance of ErlangC assuming an infinite queue capacity.

Parameters

`arrivalRate` the arrival rate.

`serviceRate` the service rate.

Methods

```
public double getProbDelay (int server)
```

Returns the delay probability : $Pr\{W > 0\}$, such that the call will wait.

Parameter

`server` the number of servers.

Returns the delay probability.

```
public static double getProbDelay (double arrivalRate, double serviceRate,  
                                   int server)
```

Returns the delay probability : $Pr\{W > 0\}$, such that the call will wait. Assumes an infinite queue.

Parameters

`arrivalRate` the arrival rate.

`serviceRate` the service rate.

`server` the number of servers.

Returns the delay probability.

```
public static double getProbDelay (double arrivalRate, double serviceRate,  
                                   int capacity, int server)
```

Returns the delay probability : $Pr\{W > 0\}$, such that the call will wait.

Parameters

`arrivalRate` the arrival rate.

`serviceRate` the service rate.

`capacity` the capacity of the queue, sets to `Integer.MAX_VALUE` for infinite queue capacity.

`server` the number of servers.

Returns the delay probability.

```
public static double[][] getStateProbDist (double arrivalRate, double  
                                           serviceRate, int capacity, int  
                                           server)
```

Returns the mass probability distribution of the states (number of calls) in the queueing system. First element is the state, the second element is the mass probability.

Parameters

`arrivalRate` the arrival rate.

`serviceRate` the service rate.

`capacity` the capacity of the queue, it must be finite and greater or equal to 0.

`server` the number of servers.

Returns the mass probability distribution or null if it cannot be computed. The first element is the state, the second element is the mass probability. The size of the vector is : number of server + capacity + 1 (for the empty state).

```
public double getServiceLevel (int server, double awt)
```

Returns the service level which is the proportion of calls that have waited less or equal to awt , $Pr\{W \leq awt\}$. awt must be given in the same unit as the arrival and service rates.

Parameters

`server` the number of servers.

`awt` the acceptable waiting time.

Returns the service level.

```
public static double getServiceLevel (double arrivalRate, double
                                     serviceRate, int server, double awt)
```

Returns the service level which is the proportion of calls that have waited less or equal to awt , $Pr\{W \leq awt\}$. awt must be given in the same unit as the arrival and service rates. This method assumes a queue with an infinite capacity.

Parameters

`arrivalRate` the arrival rate.

`serviceRate` the service rate.

`server` the number of servers.

`awt` the acceptable waiting time.

Returns the service level.

```
public static double getServiceLevel (double arrivalRate, double
                                     serviceRate, int capacity, int
                                     server, double awt)
```

Returns the service level which is the proportion of calls that have waited less or equal to awt , $Pr\{W \leq awt\}$. awt must be given in the same unit as the arrival and service rates. Give a capacity of `Integer.MAX_VALUE` for a queue with an infinite capacity.

Parameters

`arrivalRate` the arrival rate.

`serviceRate` the service rate.

`capacity` the capacity of the waiting queue.

`server` the number of servers.

`awt` the acceptable waiting time.

Returns the service level.

```
public int minServer (double awt, double sl)
```

Returns the minimum number c of servers needed to have a service level of at least sl , that is : $\min_{c \geq 0} \{c : Pr\{W \leq awt\} \geq sl\}$. This function uses a binary search.

Parameters

`awt` the acceptable waiting time.

`sl` the target service level, it must be in the interval $[0, 1]$.

Returns the minimum number of servers needed to satisfy a service level of sl .

```
public static int minServer (double arrivalRate, double serviceRate,
                             double awt, double sl)
```

Returns the minimum number c of servers needed to have a service level of at least sl , that is : $\min_{c \geq 0} \{c : Pr\{W \leq awt\} \geq sl\}$. The capacity of the queue is assumed infinite. This function uses a binary search.

Parameters

`arrivalRate` the exponential arrival rate.

`serviceRate` the exponential service rate.

`awt` the acceptable waiting time.

`sl` the target service level, it must be in the interval $[0, 1]$.

Returns the minimum number of servers needed to satisfy a service level of sl .

```
public static int minServer (double arrivalRate, double serviceRate, int
                             capacity, double awt, double sl)
```

Returns the minimum number c of servers needed to have a service level of at least sl , that is : $\min_{c \geq 0} \{c : Pr\{W \leq awt\} \geq sl\}$. If the capacity is `Integer.MAX_VALUE`, the capacity of the queue is assumed infinite. This function uses a binary search.

Parameters

`arrivalRate` the exponential arrival rate.

`serviceRate` the exponential service rate.

`capacity` the capacity of the queue.

`awt` the acceptable waiting time.

`sl` the target service level, it must be in the interval $[0, 1]$.

Returns the minimum number of servers needed to satisfy a service level of *sl*.

```
public double getAverageWaitTime (int server)
```

Returns the average wait time : $\mathbb{E}[W]$.

Parameter

`server` the number of servers.

Returns the average wait time.

```
public static double getAverageWaitTime (double arrivalRate, double  
                                         serviceRate, int server)
```

Returns the average wait time : $\mathbb{E}[W]$. It assumes an infinite queue capacity.

Parameters

`arrivalRate` the arrival rate.

`serviceRate` the service rate.

`server` the number of servers.

Returns the average wait time.

```
public static double getAverageWaitTime (double arrivalRate, double  
                                         serviceRate, int capacity, int  
                                         server)
```

Returns the average wait time : $\mathbb{E}[W]$.

Parameters

`arrivalRate` the arrival rate.

`serviceRate` the service rate.

`capacity` the capacity of the queue.

`server` the number of servers.

Returns the average wait time.

```
public double getAverageExcessTime (int server, double awt)
```

Returns the average excess time : $\mathbb{E}[(W - awt)^+]$. It corresponds to the average of waiting time exceeding *awt*.

Parameters

server the number of servers.

awt the acceptable waiting time.

Returns the average excess time.

```
public static double getAverageExcessTime (double arrivalRate, double
                                           serviceRate, int server, double
                                           awt)
```

Returns the average excess time : $\mathbb{E}[(W - awt)^+]$. It corresponds to the average of waiting time exceeding *awt*. It assumes an infinite queue capacity.

Parameters

arrivalRate the arrival rate.

serviceRate the service rate.

server the number of servers.

awt the acceptable waiting time.

Returns the average excess time.

```
public static double getAverageExcessTime (double arrivalRate, double
                                           serviceRate, int capacity, int
                                           server, double awt)
```

Returns the average excess time : $\mathbb{E}[(W - awt)^+]$. It corresponds to the average of waiting time exceeding *awt*.

Parameters

arrivalRate the arrival rate.

serviceRate the service rate.

capacity the capacity of the queue.

server the number of servers.

awt the acceptable waiting time.

Returns the average excess time.

```
public static void main (String[] args) throws Exception
```

Returns the service level and the delay probability of given parameters. Arguments to give : <arrival rate> <service rate> <number of servers> <awt> [<capacity>]. If the capacity is omitted, the queue is assumed to have an infinite capacity.

Package `umontreal.iro.lecuyer.contactcenters.server`

Manages the simulation of the contact's service process. The purpose of the contact center is to offer some service which is provided by a pool of servers or agents sharing the same skills.

An *agent group* i , represented by an instance of `AgentGroup`, contains $N_i(t) \in \mathbb{N}$ members at simulation time t . Among these agents, $N_{I,i}(t)$ are idle, and $N_{B,i}(t)$ are busy. Only $N_{F,i}(t) \leq N_{I,i}(t)$ agents are available to serve new contacts.

The service of a contact is divided in two steps. After communicating with a customer (first step), an agent can perform after-contact work (second step), e.g., update an account, take some notes, etc. After the first step, the contact may exit the system, or be transferred to another agent. However, the agent becomes free only after the second step (if any) is over.

By default, for better efficiency, an agent group does not contain an object for each agent, preventing the simulator from differentiating them. Individual agents can of course be simulated by creating groups with a single member, but regrouping the agents can be useful for more efficient routing. The subclass `DetailedAgentGroup` offers an implementation where each individual agent is a separate object with its own characteristics. Each such agent can be added to or removed from a group at any time during a simulation.

This package also provides helper classes to assign service times to contacts and compute the integrals of the number of agents over simulation time.

AgentGroup

Represents a group i of agents capable of serving some types of contacts. An instance of this class keeps counters for the number of agents in a group, and provides logic to manage the service of contacts. It also defines a list of observers being notified when the agent group changes.

An agent group contains $N_i(t) \in \mathbb{N}$ members at simulation time t . Among these agents, $N_{I,i}(t)$ are idle and $N_{B,i}(t)$ are busy. Since agents terminate their service before they leave, we can have $N_i(t) < N_{B,i}(t)$, in which case $N_{G,i}(t) = N_{B,i}(t) - N_i(t)$ *ghost agents* need to disappear after they finish their work. As a result, the true number of agents in a group i at time t is given by $N_i(t) + N_{G,i}(t)$. New contacts are not accepted by the group when $N_i(t) \leq N_{B,i}(t)$. Since $N_{B,i}(t)$ includes the ghost agents, we have

$$N_i(t) + N_{G,i}(t) = N_{B,i}(t) + N_{I,i}(t). \quad (2)$$

Some idle agents may be unavailable to serve contacts at some times during their shift. They can be taking unplanned breaks, going to the bathroom, etc. These details can be modeled in the simulation if the appropriate information is available. But in practice they are often approximated by various models such as an efficiency factor $\epsilon_i \in [0, 1]$, which corresponds to the fraction of agents being effectively busy or available to serve contacts. If $N_{B,i}(t) = 0$, the number of free agents $N_{F,i}(t)$ available to serve contacts is given by $N_{F,i}(t) = \text{round}(\epsilon_i N_i(t))$ where $\text{round}(\cdot)$ rounds its argument to the nearest integer. If $N_{B,i}(t) > 0$, the number of busy members of the group, $N_{B,i}(t) - N_{G,i}(t)$, needs to be subtracted to get $N_{F,i}(t)$. This yields:

$$\text{round}(\epsilon_i N_i(t)) + N_{G,i}(t) = N_{B,i}(t) + N_{F,i}(t). \quad (3)$$

If $\epsilon_i = 1$, $N_{F,i}(t) = N_{I,i}(t)$ and we are back to (2). This elementary efficiency model is provided because it can be used without simulating individual agents. When agents are differentiated, other more complex and more realistic models can easily be implemented by manipulating the state of agents during simulation.

The service of a contact, started by the `serve(Contact)` method, is divided in two steps. After communicating with a customer (first step), an agent can perform after-contact work (second step), e.g., update an account, take some notes, etc. After the first step, the contact may exit the system, or be transferred to another agent. However, the agent becomes free only after the second step (if any) is over. The end of these steps is scheduled using a simulation event `EndServiceEvent` that contains additional information about the service. Service can be terminated automatically through the event or manually through the `endContact(EndServiceEvent, int)` and `endService(EndServiceEvent, int)` methods of this class. Special indicators called the *end-contact type* and *end-service type* tell us which type of termination has occurred for each step. By default, the two steps of the service are terminated automatically after durations obtained using the `Contact.getDefaultContactTime()` and `Contact.getDefaultAfterContactTime()` methods of the concerned contact, respectively. These default times can be set to infinity if services need to be terminated manually, conditional on some event. The way times are obtained can also be changed

by setting value generators using `setContactTimeGenerator (int, ValueGenerator)`, and `setAfterContactTimeGenerator (int, ValueGenerator)`, or by overriding `getContactTime (EndServiceEvent)` and `getAfterContactTime (EndServiceEvent)`.

Registered *agent-group listeners* can be notified when $N_i(t)$ changes, when a service starts, and when it ends.

Note: the `AgentGroupListener` implementations are notified in the order of the list returned by `getAgentGroupListeners()`, and an agent-group listener modifying the list of listeners by using `addAgentGroupListener (AgentGroupListener)` or `removeAgentGroupListener (AgentGroupListener)` could result in unpredictable behavior.

An agent group can also be viewed as a collection of end-service events. For this reason, this class implements the `Collection` interface. The collection contains end-service events corresponding to in-progress services. Its size thus always corresponds to the number of busy agents.

```
package umontreal.iro.lecuyer.contactcenters.server;
```

```
public class AgentGroup extends AbstractCollection<EndServiceEvent>
    implements PeriodChangeListener, Initializable, Named
```

Fields

```
protected int ecTypeRet
```

The end-contact type associated with the contact time returned by `getContactTime (EndServiceEvent)`.

```
protected int esTypeRet
```

The end-service type associated with the after-contact time returned by `getAfterContactTime (EndServiceEvent)`.

Constructors

```
public AgentGroup (int n)
```

Constructs a new agent group with `n` available agents.

Parameter

`n` the number of agents in the group.

```
public AgentGroup (PeriodChangeEvent pce, int[] ns)
```

Constructs a new agent group with the period-change event `pce`, and `ns[p]` agents in the period `p`. The agent group is automatically added to the period-change event for the number of agents to be set automatically during the simulation.

Parameters

pce the period-change event defining the simulation periods.

ns the number of agents in the group for each period.

Throws

IllegalArgumentException if there is not a number of agent for each period.

Methods

```
public PeriodChangeEvent getPeriodChangeEvent()
```

Returns the period-change event associated with this agent group.

Returns the associated period-change event.

```
public double getEfficiency()
```

Returns ϵ_i , the fraction of free and busy agents available to serve contacts over the total number of agents. The default efficiency is set to 1.

Returns the agents' efficiency.

```
public void setEfficiency (double eff)
```

Changes the agents' efficiency to **eff**. This calls **setNumAgents (int)** to update the number of free agents according to the new efficiency factor. If there is no busy agent, the number of free agents is given by **getNumAgents()*eff**, rounded to the nearest integer. The efficiency factor must be in $[0, 1]$, otherwise an exception is thrown.

Parameter

eff the new efficiency.

Throws

IllegalArgumentException if the efficiency factor is smaller than 0 or greater than 1.

```
public ValueGenerator getContactTimeGenerator (int ecType)
```

Returns the value generator used to generate contact times for end-contact type **ecType**. This returns **null** if there is no value generator associated with this type of contact termination. By default, a non-null value is returned for **ecType = 0** only.

Parameter

ecType the queried end-contact type.

Returns the value generator associated with this end-contact type.

```
public ValueGenerator getAfterContactTimeGenerator (int esType)
```

Returns the value generator used to generate after-contact times for end-service type **esType**. This returns **null** if there is no value generator associated with this type of service termination.

Parameter

esType the queried end-service type.

Returns the value generator associated with this end-service type.

```
public void setContactTimeGenerator (int ecType, ValueGenerator cgen)
```

Sets the contact time generator for end-contact type **ecType** to **cgen**.

Parameters

ecType the affected end-contact type.

cgen the new contact time generator associated with this end-contact type.

Throws

IllegalArgumentException if the end-contact type is negative.

```
public void setAfterContactTimeGenerator (int esType, ValueGenerator acgen)
```

Sets the after-contact time generator for end-service type **esType** to **acgen**.

Parameters

esType the affected end-service type.

acgen the new after-contact time generator associated with this end-service type.

Throws

IllegalArgumentException if the end-service type is negative.

```
public void init()
```

Initializes the agent group for a new simulation replication. It must be called after the simulator is initialized and before it is started.

```
public boolean isKeepingEndServiceEvents()
```

Determines if this object keeps track of the end-service events for contacts in service by an agent. If this returns **true**, the events are stored. Otherwise (the default), they are stored in the SSJ event list only.

Returns the value of the keep end-service events flag.

```
public void setKeepingEndServiceEvents (boolean keepEsev)
```

Sets the keep end-service-event indicator to **keepEsev**.

Parameter

keepEsev the new value of the indicator.

See also `isKeepingEndServiceEvents()`

```
public Iterator<EndServiceEvent> endServiceEventsIterator  
( )
```

Constructs and returns an iterator for the end-service events. If `isKeepingEndServiceEvents()` returns `true`, the iterator is constructed from the set returned by `getEndServiceEvents()`. Otherwise, an illegal state exception is thrown.

Returns the iterator for end-service events.

```
public Set<EndServiceEvent> getEndServiceEvents()
```

Returns a reference to a set containing all the end-service events for this agent group. This set contains the end-service events for each contact currently served by an agent. As soon as a contact ends its service (including after-contact work), it is removed from the set. If the agent group does not keep track of these events (the default), this throws an `IllegalStateException`.

Returns the set of end-service events.

Throws

`IllegalStateException` if the agent group does not keep end-service events.

```
public EndServiceEvent serve (Contact contact)
```

Begins the service of the contact `contact` and returns the constructed end-service event. If no agent is available to serve the contact, an `IllegalStateException` is thrown. Otherwise, a contact time is obtained using `getContactTime (EndServiceEvent)`. The end-service event is then constructed and scheduled if the contact time is not infinite. If an infinite contact time is generated, one must manually abort the communication using `endContact (EndServiceEvent, int)` or schedule the end-service event. When the communication is over, the same rules are applied for generating the after-contact time using `getAfterContactTime (EndServiceEvent)`. When the after-contact time is finite, the end-service event is scheduled a second time for the service termination.

Parameter

`contact` the contact to be served.

Returns a reference to the end-service event.

Throws

`IllegalStateException` if no free agent is available.

```
public EndServiceEvent serve (Contact contact, double contactTime, int  
ecType)
```

This is similar to `serve (Contact)`, except that the specified contact time and end-contact type are used instead of generated ones. The after-contact time is generated as in `serve (Contact)`. The main purpose of this method is for recreating an end-service event based on saved state information.

Parameters

`contact` the contact being served.

`contactTime` the communication time of the contact with the agent.

`ecType` the end-contact type.

Returns the end-service event representing the service.

Throws

`IllegalStateException` if no free agent is available.

```
public EndServiceEvent serve (Contact contact, double contactTime, int  
                             ecType, double afterContactTime, int esType)
```

This is similar to `serve (Contact)` except that the contact and after-contact times are specified explicitly. The main purpose of this method is for recreating an end-service event based on saved state information.

Parameters

`contact` the contact being served.

`contactTime` the contact time.

`ecType` the end-contact type.

`afterContactTime` the after-contact time.

`esType` the end-service type.

Returns the end-service event representing the service.

Throws

`IllegalStateException` if no free agent is available.

```
public EndServiceEvent serve (EndServiceEvent oldEndServiceEvent)
```

Starts the service of a contact based on information stored in the old end-service event `oldEndServiceEvent`. If the event contains information about the effective end-contact time, i.e., if `EndServiceEvent.contactDone()` returns `true`, the method uses the effective end-contact time and end-contact type, and the scheduled end-service time and end-service type to start the service. Otherwise, it uses the scheduled end-contact time and end-contact type only.

Parameter

`oldEndServiceEvent` the old end-service event.

Returns the new end-service event.

```
public AgentGroupState save()
```

Constructs a new `AgentGroupState` instance holding the state of this agent group. The method `isKeepingEndServiceEvents()` must return `true` for this method to be called, because the state includes every contact served by agents in this group.

Returns the state of this agent group.

```
public void restore (AgentGroupState state)
```

Restores the state of this agent group by using the `restore` method of `state`.

Parameter

`state` the saved state of this agent group.

```
protected double getContactTime (EndServiceEvent es)
```

Generates and returns the contact time for the service represented by `es`. The method returns the generated value and can store an end-contact type indicator in the protected field `ecTypeRet` if the default value of 0 is not appropriate.

By default, a `MinValueGenerator` is used. For each end-contact type c with an associated value generator, a contact time C_c is generated. The scheduled contact time is $C_{c^*} = \min_c \{C_c\}$, and the end-contact type is c^* .

Parameter

`es` the end-service event.

Returns the generated contact time.

```
protected double getAfterContactTime (EndServiceEvent es)
```

Generates and returns the after-contact time for the service represented by `es`. The method returns the generated value and can store an end-service type indicator in the protected field `esTypeRet` if the default value of 0 is not appropriate.

By default, a `MinValueGenerator` is used. For each end-service type c with an associated value generator, an after-contact time C_c is generated. The scheduled after-contact time is $C_{c^*} = \min_c \{C_c\}$, and the end-service type is c^* .

Parameter

`es` the end-service event.

Returns the generated after-contact time.

```
public boolean endContact (EndServiceEvent es, int ecType)
```

Aborts the communication with a contact identified by the end-service event `es`, overriding the event's end-contact type with `ecType`. Returns `true` if the operation was successful, or `false` otherwise. Note that the after-contact time is generated and after-contact work is performed. One must call `endService (EndServiceEvent, int)` after this method to completely abort the service.

Parameters

`es` the end-service event representing the service to be aborted.

`ecType` the type of communication termination.

Returns the success indicator of the operation.

```
public boolean endService (EndServiceEvent es, int esType)
```

Aborts the service of a contact identified by the end-service event **es**, overriding the event's end-service type with **esType**. Returns **true** if the operation was successful, or **false** otherwise. For this method to return **true**, the communication between the agent and the contactor must have ended. One can use **endContact (EndServiceEvent, int)** to abort the communication.

Parameters

es the end-service event representing the after-contact work to be aborted.

esType the type of service termination.

Returns the success indicator of the operation.

```
public int getNumAgents()
```

Returns the total number of agents in the agent group. It is possible that only a fraction of these agents can serve contacts.

Returns the total number of agents in the group.

```
public void setNumAgents (int n)
```

Changes the number of agents of this group to **n**. The number of free agents is computed by multiplying **n** by the efficiency factor, rounding the result to the nearest integer, and subtracting the number of busy members of the group.

Parameter

n the total number of agents.

Throws

IllegalArgumentException if the given number of agents is negative.

```
public int[] getAllNumAgents()
```

Returns the array containing the number of agents for each period. This method cannot be used unless the agent group is constructed with a period-change event.

Returns the number of agents for each period.

Throws

IllegalStateException if the per-period numbers of agents are not available.

```
public int getNumAgents (int p)
```

Returns the number of agents in period **p**. This method cannot be used unless the agent group is constructed with a period-change event.

Parameter

p the period index.

Returns the number of agents in the period.

Throws

`IllegalStateException` if the per-period numbers of agents are not available.

`ArrayIndexOutOfBoundsException` if the period index is negative or greater than or equal to the number of periods.

```
public void setNumAgents (int p, int n)
```

Sets the number of agents in period `p` to `n`. This method cannot be used unless the agent group is constructed with a period-change event.

Parameters

`p` the period index.

`n` the new number of agents.

Throws

`IllegalStateException` if the per-period numbers of agents are not available.

`ArrayIndexOutOfBoundsException` if the period index is negative or greater than or equal to the number of periods.

```
public void setNumAgents (int[] allNumAgents)
```

Sets the vector giving the number of agent for each period to `allNumAgents`.

Parameter

`allNumAgents` the new vector of agents.

```
public int getNumGhostAgents()
```

Returns $N_{G,i}(t)$, the number of agents that should disappear immediately after they have finished serving a contact. Such ghost agents appear when the total number of agents is set to be smaller than the number of busy agents.

Returns the number of ghost agents.

```
public int getNumIdleAgents()
```

Returns $N_{I,i}(t)$, the number of idle agents in this agent group. Since only a fraction of these idle agents can serve contacts, the returned value is greater than or equal to `getNumFreeAgents()`. If `getEfficiency()` returns 1, this returns the same value as `getNumFreeAgents()`.

Returns the number of idle agents.

```
public int getNumFreeAgents()
```

Returns $N_{F,i}(t)$, the total number of agents in the agent group which are available to process contacts. This number must always be smaller than or equal to the total number of agents.

Returns the number of free agents in the group.

```
public int getNumBusyAgents()
```

Returns $N_{B,i}(t)$, the number of busy agents in the group. At any time during the simulation, the value returned by this method should be smaller than or equal to the sum of `getNumAgents()` and `getNumGhostAgents()`.

Returns the number of busy agents.

```
public int getNumBusyAgents (int k)
```

Returns the number of busy agents serving contacts of type k .

Parameter

k the contact type index.

Returns the number of busy agents serving contacts of type k .

```
public int getId()
```

Returns the identifier associated with this agent group. This identifier, which defaults to -1, can be used as an index in routers.

Returns the identifier associated with this agent group.

```
public void setId (int id)
```

Sets the identifier of this agent group to id . Once this identifier is set to a positive or 0 value, it cannot be changed anymore. This method is automatically called by the router when an agent group is connected. If one tries to attach the same group to different routers, the group must have the same index for each of them. For this reason, if one tries to change the identifier, an `IllegalStateException` is thrown.

Parameter

id the new identifier associated with the agent group.

Throws

`IllegalStateException` if the identifier was already set.

```
public void addAgentGroupListener (AgentGroupListener listener)
```

Adds the agent-group listener `listener` to this object.

Parameter

`listener` the agent-group listener being added.

Throws

`NullPointerException` if `listener` is null.

```
public void removeAgentGroupListener (AgentGroupListener listener)
```

Removes the agent-group listener `listener` from this object.

Parameter

listener the agent-group listener being removed.

```
public void clearAgentGroupListeners()
```

Removes all the agent-group listeners registered with this agent group.

```
public List<AgentGroupListener> getAgentGroupListeners()
```

Returns an unmodifiable list containing all the agent-group listeners registered with this agent group.

Returns the list of all registered agent-group listeners.

```
protected void notifyInit()
```

Notifies every registered listener that this agent group has been initialized.

```
protected void notifyChange()
```

Notifies every registered listener that the number of agents of this group has changed.

```
protected void notifyBeginService (EndServiceEvent es)
```

Notifies every registered listener that a service, represented by **es**, was started by this agent group.

Parameter

es the end-service event representing the service.

```
protected void notifyEndContact (EndServiceEvent es, boolean aborted)
```

Notifies every registered listener that the communication part of the service represented by **es** has ended.

Parameters

es the end-service event.

aborted determines if the service was aborted or terminated normally.

```
protected void notifyEndService (EndServiceEvent es, boolean aborted)
```

Notifies every registered listener that the service represented by **es** is finished.

Parameters

es the end-service vent representing the ended service.

aborted determines if the after-contact work was aborted or terminated normally.

```
public Map<Object, Object> getAttributes()
```

Returns the map containing the attributes for this agent group. Attributes can be used to add user-defined information to agent group objects at runtime, without creating a subclass. However, for maximal efficiency, it is recommended to create a subclass of **AgentGroup** instead of using attributes.

Returns the map containing the attributes for this object.

EndServiceEvent

Represents the simulation event for a contact's end of service. It is constructed and returned by the `AgentGroup.serve (Contact)` method and can be used to abort the service of a contact, dynamically modify its service time, or get information about the service. The event contains scheduled as well as effective information. A scheduled information is determined at the time the event is scheduled. For example, the schedule contact time is the contact time which was generated at the beginning of the service. An effective information is determined at the time the event occurs, or the service is aborted. It is different from the scheduled information only when the service is aborted.

```
package umontreal.iro.lecuyer.contactcenters.server;

public class EndServiceEvent extends Event
    implements ContactStepInfo, Cloneable
```

Constructor

```
protected EndServiceEvent (AgentGroup group, Contact contact, double
                           beginServiceTime)
```

Constructs a new end-service event with contact `contact` served by an agent in group `group`, with service beginning at simulation time `beginServiceTime`.

This constructor is rarely used directly; the recommended way to create end-service events is to use `AgentGroup.serve (Contact)`.

Parameters

`group` the associated agent group.

`contact` the contact being served.

`beginServiceTime` the time at which the service begins.

Methods

```
public Contact getContact()
```

Returns the contact being served.

Returns the contact being served.

```
public double getBeginServiceTime()
```

Returns the simulation time at which the service started.

Returns the time of beginning of service.

```
public double getScheduledContactTime()
```

Returns the scheduled duration of the communication with the contact.

Returns the scheduled contact time.

```
public double getEffectiveContactTime()
```

Returns the effective contact time. If the communication is not terminated, this throws an `IllegalStateException`.

Returns the effective contact time.

Throws

`IllegalStateException` if the communication is not terminated.

```
public double getScheduledAfterContactTime()
```

Returns the scheduled after-contact time. If the communication is not terminated, an `IllegalStateException` is thrown.

Returns the scheduled after-contact time.

Throws

`IllegalStateException` if the communication is not terminated.

```
public double getEffectiveAfterContactTime()
```

Returns the effective after-contact time. If the service is not terminated, this throws an `IllegalStateException`.

Returns the effective after-contact time.

Throws

`IllegalStateException` if the service is not terminated.

```
public int getScheduledEndContactType()
```

Returns the type of contact termination that will occur when this event happens for the first time. This scheduled end-contact type can be overridden by using the `AgentGroup.endContact (EndServiceEvent, int)` method.

Returns the scheduled end-contact type.

```
public int getScheduledEndServiceType()
```

Returns the type of the service termination that will occur when this event happens for the second time. This scheduled end-service type can be overridden by using the `AgentGroup.endService (EndServiceEvent, int)` method.

Returns the scheduled end-service type.

```
public int getEffectiveEndContactType()
```

Returns the effective type of contact termination. If the communication is not terminated, this throws an `IllegalStateException`.

Returns the effective end-contact type.

Throws

`IllegalStateException` if the communication is not terminated.

```
public int getEffectiveEndServiceType()
```

Returns the effective type of the service termination. If the service is not terminated, this throws an `IllegalStateException`.

Returns the effective end-service type.

Throws

`IllegalStateException` if the service is not terminated.

```
public void setScheduledEndContactType (int ecType)
```

Changes the type of contact termination that will occur when this event happens to `ecType`. If the communication is terminated, this throws an `IllegalStateException`.

Parameter

`ecType` the new end-contact type.

Throws

`IllegalStateException` if the communication is terminated.

```
public void setScheduledEndServiceType (int esType)
```

Changes the type of service termination that will occur when this event happens to `esType`. If the service is terminated, this throws an `IllegalStateException`.

Parameter

`esType` the new end-service type.

Throws

`IllegalStateException` if the service is terminated.

```
public AgentGroup getAgentGroup()
```

Returns the agent group containing the agent serving the contact.

Returns the agent group serving the contact.

```
public boolean endContact (int ecType1)
```

Terminates the communication part of the service represented by this event, with end-contact type `ecType`, and returns `true` if and only if the communication part was terminated successfully. This method calls `getAgentGroup() AgentGroup.endContact (EndServiceEvent, int)`, and returns the result.

Parameter

ecType1 the end-contact type.

Returns the success indicator of the operation.

```
public boolean endService (int esType1)
```

Terminates the after-contact part of the service represented by this event, with end-service type **esType**, and returns **true** if and only if the after-contact part was terminated successfully. This method calls `getAgentGroup() AgentGroup.endService (EndServiceEvent, int)`, and returns the result.

Parameter

esType1 the end-service type.

Returns the success indicator of the operation.

```
public boolean contactDone()
```

Determines if the communication is finished between the contact and the agent.

Returns **true** if the contact was served, **false** otherwise.

```
public boolean afterContactDone()
```

Determines if the after-contact work or service is terminated by the agent.

Returns **true** if the after-contact work is done, **false** otherwise.

```
public boolean wasGhostAgent()
```

Determines if the agent ending the service of the contact disappears after the service is completed.

Returns the ghost agent status.

```
public boolean isObsolete()
```

Determines if this event is obsolete. When calling `AgentGroup.init()`, some end-service events might still be in the simulator's event list. Since this agent group does not store every scheduled end-service event by default, one must use this method in `actions()` to test if this event is obsolete. If that returns **true**, one should return immediately.

Returns **true** for an obsolete event, **false** otherwise.

```
public EndServiceEvent clone()
```

Returns a copy of this event. This method clones every field of the event, except the agent group which is not cloneable.

```
public EndServiceEvent clone (Contact clonedContact)
```

Similar to `clone()`, but initializes the contact of the cloned event with `clonedContact` instead of a clone of the contact returned by `getContact()`. This method can be useful when cloning a contact `c` for which `c.getSteps()` returns a non-empty list containing end-service events. In that case, the contact associated with the events included in `c.getSteps()` must correspond to `c` rather than clones of `c`.

DetailedAgentGroup

Extends the `AgentGroup` class for a detailed agent group, where individual agents can be differentiated. When serving a contact, a specific agent, represented by an instance of the class `Agent`, must be chosen automatically using the longest idle policy, or manually. At any time during the simulation, agents can be added to or removed from the group. Agents can also be made available or unavailable to process new contacts.

```
package umontreal.iro.lecuyer.contactcenters.server;
```

```
public class DetailedAgentGroup extends AgentGroup
```

Constructors

```
public DetailedAgentGroup (int n)
```

Constructs a new agent group with `n` available agents.

Parameter

`n` the number of agents in the group.

```
public DetailedAgentGroup (PeriodChangeEvent pce, int[] ns)
```

Constructs a new agent group with the period-change event `pce`, and `ns[p]` agents in the period `p`. The agent group is automatically added to the period-change event for the number of agents to be set automatically during the simulation.

Parameters

`pce` the period-change event defining the simulation periods.

`ns` the number of agents in the group for each period.

Throws

`IllegalArgumentException` if there is not a number of agents for each period.

Methods

```
public Simulator simulator()
```

Returns a reference to the simulator used to obtain simulation times at which agents are added or become free, for computing login and idle times of agents.

Returns the attached simulator.

```
public void setSimulator (Simulator sim)
```

Sets the attached simulator of this agent group to `sim`.

Parameter

`sim` the new attached simulator.

```
public void setNumAgents (int n)
```

Sets the number of agents in the agent group to `n`. When the number of agents is increased, the `createAgent()` method is used to create the required new agents. When removing agents, the method uses the busyness status (busy agents are removed only if there is no more idle agent), and the login time (the agent with the longest login time is chosen) to decide which agent to remove. The methods `addAgent (Agent)` and `removeAgent (Agent)` are used to add and remove the agents.

Parameter

`n` the new number of agents.

Throws

`IllegalArgumentException` if the number of agents is negative.

```
public void addAgent (Agent agent)
```

Adds the agent `agent` to the agent group. When an agent is a ghost, it can be added back to its previous group, but it cannot be added to another group until he has terminated his in-progress service. When an agent is not in any group and not serving any contact, he can be added to any group.

Parameter

`agent` the agent being added.

Throws

`NullPointerException` if `agent` is null.

`IllegalArgumentException` if the agent is already in a group.

```
public void removeAgent (Agent agent)
```

Removes the agent `agent` from this agent group. If the agent is serving a contact, it becomes a ghost until the contact is served.

Parameter

`agent` the agent being removed.

Throws

`NullPointerException` if `agent` is null.

`IllegalArgumentException` if the removed agent is not in this group.

```
public boolean isAddingAgent()
```

Determines if this agent group is currently adding an agent using the `addAgent (Agent)` method. This method can be used by `AgentGroupListener.agentGroupChange (AgentGroup)` to determine the origin of a change in an observed agent group. If an agent is added explicitly using `addAgent (Agent)`, this method returns `true`. Otherwise, the change originates from a call to `setNumAgents (int)`.

Returns the result of the test.

```
public boolean isRemovingAgent()
```

Determines if an agent is currently being removed using `removeAgent (Agent)`. This method can be used by an agent-group listener as described in `isAddingAgent()`.

Returns the result of the test.

```
public List<Agent> getIdleAgents()
```

Returns a list containing all the idle agent objects. These idle agents are not necessarily available to process contacts.

Returns the idle agents.

```
public List<Agent> getBusyAgents()
```

Returns a list containing all the busy agent objects which are members of this group. This excludes ghost agents since they have been removed from the group.

Returns the busy agents.

```
public List<Agent> getGhostAgents()
```

Returns a list containing all the ghost agent objects having been members of this agent group and finishing an in-progress service.

Returns the ghost agents.

```
public int getNumFreeAgents()
```

Returns the number of free agents available to process contacts. These are the free agents for which `Agent.isAvailable()` returns `true`.

Returns the number of free agents.

```
public double getEfficiency()
```

Returns the current efficiency of this agent group, which is given by the fraction of available agents (free or busy) over the total number of agents. This efficiency can change when the agents are added to or removed from the group, or when the availability status of agents changes.

Returns the efficiency of the agent group.

```
public void setEfficiency (double eff)
```

Sets the efficiency of the agents in the group. The method computes a target number of available agents by multiplying $N_i(t)$ by `eff` and rounding the result to the nearest integer. Some agents are then made available or unavailable to meet this target, starting with free agents, then with busy agents. This method is provided for compatibility with the `AgentGroup` base class. The recommended way to change the efficiency is to change the availability of each individual agent by using `Agent.setAvailable (boolean)`. This permits the implementation of more complex and realistic models of agents' availability.

Parameter

eff the new agent group's efficiency.

Throws

IllegalArgumentException if **eff** is smaller than 0 or greater than 1.

```
public Agent getLongestIdleAgent()
```

Returns the idle agent with the longest idle time in this agent group. If all agents are busy or unavailable to process new contacts, this returns **null**.

Returns the idle agent with the longest idle time.

```
public EndServiceEventDetailed serve (Contact contact)
```

Begins the service of the contact **contact** by the agent with the longest idle time in this group. After the agent is selected, the **serve (Contact, Agent)** method is called to begin the service.

Parameter

contact the contact being served.

Returns the end of service event being created.

```
public EndServiceEventDetailed serve (Contact contact, Agent agent)
```

Begins the service of the contact **contact** by the agent **agent**. Returns the constructed end-service event. Communication times are generated using **getContactTime (EndServiceEvent)**, and after-contact times are obtained using **getAfterContactTime (EndServiceEvent)**.

Parameters

contact the contact being served.

agent the agent serving the contact.

Returns the constructed end-service event.

Throws

NullPointerException if an argument is **null**.

IllegalArgumentException if the given agent is in the wrong agent group.

IllegalStateException if the agent is not available or already serving a contact.

```
public EndServiceEventDetailed serve (Contact contact, Agent agent, double  
                                     contactTime, int ecType)
```

This is similar to **serve (Contact, Agent)**, except that the specified contact time and end-contact type are used instead of generated ones. The after-contact time is generated as in **serve (Contact)**. The main purpose of this method is for recreating an end-service event based on saved state information.

Parameters

`contact` the contact being served.

`agent` the agent serving the contact.

`contactTime` the communication time of the contact with the agent.

`ecType` the end-contact type.

Returns the end-service event representing the service.

Throws

`IllegalStateException` if no free agent is available.

```
public EndServiceEventDetailed serve (Contact contact, Agent agent, double
                                     contactTime, int ecType, double
                                     afterContactTime, int esType)
```

This is similar to `serve (Contact, Agent)` except that the contact and after-contact times are specified explicitly. The main purpose of this method is for recreating an end-service event based on saved state information.

Parameters

`contact` the contact being served.

`agent` the agent serving the contact.

`contactTime` the contact time.

`ecType` the end-contact type.

`afterContactTime` the after-contact time.

`esType` the end-service type.

Returns the end-service event representing the service.

Throws

`IllegalStateException` if no free agent is available.

```
protected double getContactTime (EndServiceEvent es)
```

By default, this method calls `Agent.getContactTime (Contact)`. If this returns `Double.NaN`, the method of the superclass is called.

Parameter

`es` the end-service event.

Returns the generated contact time.

```
protected double getAfterContactTime (EndServiceEvent es)
```

By default, this method calls `Agent.getAfterContactTime (Contact)`. If the called method returns `Double.NaN`, the method calls the equivalent method of the superclass.

Parameter

es the end-service event.

Returns the generated after-contact time.

protected Agent createAgent()

Constructs a new agent object. This method can be overridden to create subclasses of **Agent** containing additional information.

Returns the constructed agent object.

EndServiceEventDetailed

Represents the end-service event for a detailed agent group.

```
package umontreal.iro.lecuyer.contactcenters.server;  
  
public class EndServiceEventDetailed extends EndServiceEvent
```

Constructor

```
protected EndServiceEventDetailed (Contact contact, Agent agent, double  
                                   beginServiceTime)
```

Constructs a new end-service event with contact **contact** served by agent **agent**, with service beginning at **beginServiceTime**.

This constructor is rarely used directly; the recommended way to create end-service events is to use **DetailedAgentGroup.serve (Contact)**.

Parameters

contact the contact being served.

agent the agent serving the contact.

beginServiceTime the simulation at which the service begins.

Method

```
public Agent getAgent()
```

Returns the agent serving or having served the contact.

Returns the serving agent.

Agent

Represents an individual agent in a detailed agent group.

Note: the `AgentListener` implementations are notified in the order of the list returned by `getAgentListeners()`, and an agent listener modifying the list of listeners by using `addAgentListener (AgentListener)` or `removeAgentListener (AgentListener)` could result in unpredictable behavior.

```
package umontreal.iro.lecuyer.contactcenters.server;
```

```
public class Agent implements Initializable, Named
```

Fields

```
protected int ecType
```

The end-contact type associated with the contact time returned by `getContactTime (Contact)`.

```
protected int esType
```

The end-service type associated with the after-contact time returned by `getAfterContactTime (Contact)`.

Methods

```
public AgentState save()
```

Constructs and returns a token object containing the state of this agent.

Returns the state of this agent.

```
public void restore (AgentState state)
```

Restores the state of this agent by using the given state object `state`.

Parameter

`state` the state of the agent.

```
public void addAgentListener (AgentListener listener)
```

Adds the agent listener `listener` to this object.

Parameter

`listener` the agent listener being added.

Throws

`NullPointerException` if `listener` is `null`.

```
public void removeAgentListener (AgentListener listener)
```

Removes the agent listener `listener` from this object.

Parameter

`listener` the agent listener being removed.

```
public void clearAgentListeners()
```

Removes all the agent listeners registered with this agent.

```
public List<AgentListener> getAgentListeners()
```

Returns an unmodifiable list containing all the agent listeners registered with this agent.

Returns the list of all registered agent listeners.

```
public EndServiceEventDetailed serve (Contact contact)
```

Instructs this agent to begin the service of the contact `contact`, and returns the constructed end-service event representing the service. This method calls `getAgentGroup().serve(contact, this)`.

Parameter

`contact` the contact to be served.

Returns the end-service event representing the service.

```
public ValueGenerator getContactTimeGenerator (int ecType1)
```

Returns the value generator used to generate contact times for end-contact type `ecType`. This returns `null` if there is no value generator associated with this type of contact termination.

Parameter

`ecType1` the queried end-contact type.

Returns the value generator associated with this end-contact type.

```
public ValueGenerator getAfterContactTimeGenerator (int esType1)
```

Returns the value generator used to generate after-contact times for end-service type `esType`. This returns `null` if there is no value generator associated with this type of service termination.

Parameter

`esType1` the queried end-service type.

Returns the value generator associated with this end-service type.

```
public void setContactTimeGenerator (int ecType, ValueGenerator cgen)
```

Sets the contact time generator for end-contact type `ecType` to `cgen`.

Parameters

`ecType` the affected end-contact type.

`cgen` the new contact time generator associated with this end-contact type.

Throws

`IllegalArgumentException` if the end-contact type is negative.

```
public void setAfterContactTimeGenerator (int esType, ValueGenerator acgen)
```

Sets the after-contact time generator for end-service type `esType` to `acgen`.

Parameters

`esType` the modified end-service type.

`acgen` the new after-contact time generator associated with this end-service type.

Throws

`IllegalArgumentException` if the end-service type is negative.

```
public void init()
```

Initializes this agent for a new simulation replication.

```
public boolean isAvailable()
```

Determines if the agent is available, or is serving contacts.

Returns the availability status of this agent.

```
public void setAvailable (boolean avail)
```

Sets the availability status of this agent to `avail`. If this method is called with `true`, the agent will be capable of processing new contacts (the default). Otherwise, it will not receive new contacts. This does not affect the contact being served by this agent if it is busy.

Parameter

`avail` the new availability status of this agent.

```
public boolean isGhost()
```

Determines if this agent is a ghost, i.e., if it was removed from an agent group before it has ended the service of a contact.

Returns `true` if the agent is a ghost agent, `false` otherwise.

```
public boolean isBusy()
```

Determines if this agent is busy.

Returns the agent's busyness indicator.

```
public EndServiceEventDetailed getEndServiceEvent()
```

Returns the current end-service event for this agent, or `null` if the agent is not busy.

Returns the current end-service event, or `null`.

```
public double getIdleSimTime()
```

Returns the last simulation time at which this agent became idle.

Returns the simulation idle time of this agent.

Throws

`IllegalStateException` if this agent is not idle.

```
public double getIdleTime()
```

Returns the time elapsed since the last moment this agent became idle. This corresponds to the current simulation time minus the result of `getIdleSimTime()`.

Returns the agent's idle time.

```
public double getFirstLoginTime()
```

Returns the first simulation time at which this agent was added to an agent group.

Returns the agent's first login time.

```
public double getLastLoginTime()
```

Returns the last simulation time at which this agent was added to an agent group.

Returns the agent's last login time.

```
public DetailedAgentGroup getAgentGroup()
```

Returns the detailed agent group this agent is part of, or `null` if the agent is not in a group.

Returns the parent agent group.

```
protected double getContactTime (Contact contact)
```

Generates and returns the contact time associated with the contact `contact`. The method returns the generated value and can store an end-contact type indicator in the protected field `ecType` if the default value of 0 is not appropriate. If this returns `Double.NaN`, the contact time will be generated by the parent agent group.

By default, a `MinValueGenerator` is used. For each end-contact type c with an associated value generator, a contact time C_c is generated. The scheduled contact time is $C_{c^*} = \min_c \{C_c\}$, and the end-contact type is c^* .

Parameter

`contact` the contact being served.

Returns the generated contact time.

`protected double getAfterContactTime (Contact contact)`

Generates and returns the after-contact time associated with the contact `contact`. The method returns the generated value and can store an end-service type indicator in the protected field `esType` if the default value of 0 is not appropriate. If this returns `Double.NaN`, the after-contact time will be generated by the parent agent group.

By default, a `MinValueGenerator` is used. For each end-service type c with an associated value generator, an after-contact time C_c is generated. The scheduled after-contact time is $C_{c^*} = \min_c \{C_c\}$, and the end-service type is c^* .

Parameter

`contact` the contact being served.

Returns the generated after-contact time.

`public int getId()`

Returns the identifier associated with this agent. This identifier, which defaults to -1, can be used as an index in routers.

Returns the identifier associated with this agent.

`public void setId (int id)`

Sets the identifier of this agent to `id`. Once this identifier is set to a positive or 0 value, it cannot be changed anymore.

Parameter

`id` the new identifier associated with the agent.

Throws

`IllegalStateException` if the identifier was already set.

`public Map<Object, Object> getAttributes()`

Returns the map containing the attributes for this agent. Attributes can be used to add user-defined information to agent objects at runtime, without creating a subclass. However, for maximal efficiency, it is recommended to create a subclass of `Agent` instead of using attributes.

Returns the map containing the attributes for this object.

AgentGroupSet

Represents a set of agent groups for which it is possible to get the total number of members.

```
package umontreal.iro.lecuyer.contactcenters.server;
```

```
public class AgentGroupSet extends AbstractSet<AgentGroup>  
    implements Initializable, Named, Cloneable
```

Methods

```
public int getNumAgents()
```

Returns the total number of agents currently in the registered agent groups.

Returns the total number of agents.

```
public int getNumFreeAgents()
```

Returns the total number of free agents currently in the set of agent groups.

Returns the total number of free agents.

```
public int getNumBusyAgents()
```

Returns the total number of busy agents currently in the set of agent groups.

Returns the total number of busy agents.

```
public int getNumIdleAgents()
```

Returns the total number of idle agents currently in the set of agent groups.

Returns the total number of idle agents.

```
public int getNumGhostAgents()
```

Returns the total number of ghost agents currently in the set of agent groups.

Returns the total number of ghost agents.

```
public boolean add (AgentGroup group)
```

Adds the agent group `group` to this set of agent groups.

Parameter

`group` the agent group being added.

Throws

`NullPointerException` if `group` is null.

```
public boolean remove (Object group)
```

Removes the agent group `group` from this set of agent groups.

Parameter

`group` the agent group being removed.

Throws

`NullPointerException` if `group` is null.

```
public void clear()
```

Removes all the agent groups contained in this set of agent groups.

```
public void init()
```

Initializes all the agent groups in this set of agent groups.

```
public void initStat()
```

Initializes the statistical collectors for this set of agent groups. If statistical collecting is turned OFF, this throws an `IllegalStateException`.

Throws

`IllegalStateException` if statistical collecting is turned OFF.

```
public boolean isStatCollecting()
```

Determines if this set of agent groups is collecting statistics about the number of agents. If this returns `true`, statistical collecting is turned ON. Otherwise (the default), it is turned OFF.

Returns the state of statistical collecting.

```
public void setStatCollecting (boolean b)
```

Sets the state of statistical collecting to `b`. If `b` is `true`, statistical collecting is turned ON, and the statistical collectors are created or reinitialized. If `b` is `false`, statistical collecting is turned OFF.

Parameter

`b` the new state of statistical collecting.

```
public void setStatCollecting (Simulator sim)
```

Enables statistical collecting, and uses the given simulator `sim`. The simulator is used by the internal accumulates when the simulation time is required to update probes with new values.

Parameter

`sim` the simulator attached to accumulates.

```
public Accumulate getStatNumAgents()
```

Returns the statistical collector for the number of agents in the agent groups. This returns a non-null value only if statistical collecting was turned ON since this object was constructed.

Returns the statistical collector for the number of agents.

```
public Accumulate getStatNumFreeAgents()
```

Returns the statistical collector for the number of free agents in the agent groups. This returns a non-null value only if statistical collecting was turned ON since this object was constructed.

Returns the statistical collector for the number of free agents.

```
public Accumulate getStatNumBusyAgents()
```

Returns the statistical collector for the number of busy agents in the agent groups. This returns a non-null value only if statistical collecting was turned ON since this object was constructed.

Returns the statistical collector for the number of busy agents.

```
public Accumulate getStatNumIdleAgents()
```

Returns the statistical collector for the number of idle agents in the agent groups. This returns a non-null value only if statistical collecting was turned ON since this object was constructed.

Returns the statistical collector for the number of idle agents.

```
public Accumulate getStatNumGhostAgents()
```

Returns the statistical collector for the number of ghost agents in the agent groups. This returns a non-null value only if statistical collecting was turned ON since this object was constructed.

Returns the statistical collector for the number of ghost agents.

AgentGroupListener

Represents an agent-group listener which is notified when the number of agents in a group is modified or when a service starts or ends.

```
package umontreal.iro.lecuyer.contactcenters.server;
```

```
public interface AgentGroupListener
```

Methods

```
public void agentGroupChange (AgentGroup group)
```

This method is called when the number of available or free agents in the agent group `group` is changed. This happens when the `AgentGroup.setNumAgents (int)` method is called, or when the efficiency is changed. This is also called when `DetailedAgentGroup.addAgent (Agent)` or `DetailedAgentGroup.removeAgent (Agent)` are used.

Parameter

`group` the agent group being modified.

```
public void beginService (EndServiceEvent ev)
```

This method is called after the service of a contact by an agent was started. The end-service event `ev` holds all the available information about the service.

Parameter

`ev` the end-service event associated with the contact being served.

```
public void endContact (EndServiceEvent ev)
```

This method is called after the communication of a contact with an agent was terminated, with `ev` containing all the information.

Parameter

`ev` the end-service event associated with the served contact.

```
public void endService (EndServiceEvent ev)
```

This method is called after the service of a contact by an agent was terminated. The service includes the communication as well as the after-contact work.

Parameter

`ev` the end-service event associated with the served contact.

```
public void init (AgentGroup group)
```

This method is called after the `AgentGroup.init()` method is called for the agent group `group`.

Parameter

`group` the agent group being initialized.

AgentListener

Represents an agent listener being notified when the state of an individual agent changes.

```
package umontreal.iro.lecuyer.contactcenters.server;
```

```
public interface AgentListener
```

Methods

```
public void agentAvailable (Agent agent, boolean avail)
```

This method is called when the availability status of the agent `agent` changes to `avail`.

Parameters

`agent` the agent being affected.

`avail` the new availability status.

```
public void agentAdded (Agent agent, DetailedAgentGroup group)
```

This method is called when the agent `agent` is added to the agent group `group`.

Parameters

`agent` the agent being added.

`group` the agent group the agent is added to.

```
public void agentRemoved (Agent agent, DetailedAgentGroup group)
```

This method is called when the agent `agent` is removed from the agent group `group`.

Parameters

`agent` the agent being removed.

`group` the agent group the agent is removed from.

```
public void init (Agent agent)
```

This method is called when the `Agent.init()` method is called.

Parameter

`agent` the initialized agent.

```
public void beginService (EndServiceEventDetailed ev)
```

This method is called after the service of a contact by an agent is started. The end-service event `ev` holds all the available information about the service.

Parameter

ev the end-service event associated with the contact being served by an agent.

```
public void endContact (EndServiceEventDetailed ev)
```

This method is called when the communication with a contact is terminated. The end-service event **ev** holds all the available information about the service.

Parameter

ev the end-service event associated with the served contact.

```
public void endService (EndServiceEventDetailed ev)
```

This method is called after the service of a contact by an agent was terminated. The service includes the communication and the after-contact work. The end-service event **ev** holds all the available information about the service.

Parameter

ev the end-service event associated with the served contact.

GroupVolumeStat

Computes statistics for a specific agent group. Using accumulates, this class can compute integrals of $N_i(t)$, $N_{I,i}(t)$, $N_{B,i}(t)$, $N_{G,i}(t)$, and $N_{F,i}(t)$, for agent group i from the last call to `init()` to the current simulation time. Optionally, it can also compute the integral for $N_{B,i,k}(t)$, the number of busy agents in group i serving contacts of type k , for $k = 0, \dots, K-1$.

```
package umontreal.iro.lecuyer.contactcenters.server;
```

```
public class GroupVolumeStat implements Cloneable
```

Constructors

```
public GroupVolumeStat (AgentGroup group)
```

Constructs a new agent-group volume statistical probe observing the agent group `group` and only computing aggregate statistics. This is equivalent to `GroupVolumeStat (group, 0)`.

Parameter

`group` the observed agent group.

```
public GroupVolumeStat (AgentGroup group, int numTypes)
```

Constructs a new agent-group volume statistical probe observing the agent group `group`, and supporting `numTypes` contact types.

Parameters

`group` the observed agent group.

`numTypes` the number of contact types.

Throws

`IllegalArgumentException` if `numTypes` is negative.

Methods

```
public void setSimulator (Simulator sim)
```

Sets the simulator attached to internal accumulates to `sim`.

Parameter

`sim` the new simulator.

Throws

`NullPointerException` if `sim` is null.

```
public final AgentGroup getAgentGroup()
```

Returns the agent group currently associated with this object.

Returns the currently associated agent group.

```
public final void setAgentGroup (AgentGroup agentGroup)
```

Sets the associated agent group to `agentGroup`. If the given group is `null`, the statistical collector is disabled until a non-`null` agent group is given. This can be used during a replication if the integrals must be computed during some periods only.

Parameter

`agentGroup` the new associated agent group.

```
public Accumulate getStatNumAgents()
```

Returns the statistical probe computing the integral of the total number of agents over the simulation time.

Returns the statistical probe for the total number of agents.

```
public Accumulate getStatNumGhostAgents()
```

Returns the statistical probe computing the integral of the number of ghost agents over the simulation time.

Returns the statistical probe for the number of ghost agents.

```
public Accumulate getStatNumIdleAgents()
```

Returns the statistical probe computing the integral of the number of idle (available and unavailable) agents over the simulation time.

Returns the statistical probe for the number of idle agents.

```
public Accumulate getStatNumFreeAgents()
```

Returns the statistical probe computing the integral of the number of free agents over the simulation time.

Returns the statistical probe for the number of free agents.

```
public Accumulate getStatNumBusyAgents()
```

Returns the statistical probe computing the integral of the number of busy agents over the simulation time.

Returns the statistical probe for the number of busy agents.

```
public Accumulate getStatNumBusyAgents (int k)
```

Returns the statistical probe computing the integral of the number of busy agents serving contacts of type `k`, over the simulation time.

Parameter

`k` the queried contact type.

Returns the service volume statistical probe.

Throws

ArrayIndexOutOfBoundsException if *k* is negative or greater than or equal to the number of supported contact types.

```
public int getNumContactTypes()
```

Returns the number of contact types supported by this object.

Returns the number of supported contact types.

```
public GroupVolumeStat clone()
```

Constructs and returns a clone of this agent-group statistical collector. This method clones the internal statistical collectors, but the clone has no associated agent group. This can be used to save the state of the statistical collector for future restoration.

Returns a clone of this object.

GroupVolumeStatMeasureMatrix

Agent group statistical collector implementing `MeasureMatrix`. This class extends `GroupVolumeStat` and implements the `MeasureMatrix` interface and defines measures for the service, idle, working, and total volumes. The service volume corresponds to the integral of the number of busy agents $N_{B,i}(t)$ obtained by `AgentGroup.getNumBusyAgents()`. The idle volume is the integral of the number of idle agents $N_{I,i}(t)$ over the simulation time. This is obtained using `AgentGroup.getNumIdleAgents()`. The working volume is the integral of the number of working agents, $N_{B,i}(t) + N_{F,i}(t)$ over the simulation time, obtained by `AgentGroup.getNumBusyAgents()`, and `AgentGroup.getNumFreeAgents()`. The total volume corresponds to the integral of the number of agents $\int_0^T (N_i(t) + N_{G,i}(t))dt = \int_0^T (N_{B,i}(t) + N_{I,i}(t))dt$ over the simulation time. This quantity is given by the sum of the accumulates returned by the methods `AgentGroup.getNumAgents()` and `AgentGroup.getNumGhostAgents()`. These quantities can be used to compute the agent group's occupancy ratio, which is the ratio of the service volume and total volume, or the ratio of the service volume over the working volume.

This class can be given the number of contact types K to track for computing $\int_0^T N_{B,i,k}(t) dt$. If $K > 1$, the measure $0 \leq k < K$ represents the integral of the number of busy agents serving contacts of type k over the simulation time. Measures K through $K + 3$ represents respectively the service, idle, working, and total volumes.

When $K = 1$, the measure 0 corresponds to the service volume, the measure 1, to the idle volume, and the measure 2 is the working volume, and measure 3 is the total volume.

Since this measure matrix supports only one period, it must be combined with `IntegralMeasureMatrix` for one to get the measures for each period.

```
package umontreal.iro.lecuyer.contactcenters.server;
```

```
public class GroupVolumeStatMeasureMatrix extends GroupVolumeStat
    implements MeasureMatrix
```

Constructors

```
public GroupVolumeStatMeasureMatrix (AgentGroup group)
```

Constructs a new agent-group volume statistical probe observing the agent group `group` and only computing aggregate statistics. This is equivalent to `GroupVolumeStat (group, 0)`.

Parameter

`group` the observed agent group.

```
public GroupVolumeStatMeasureMatrix (Simulator sim, AgentGroup group)
```

Equivalent to `GroupVolumeStatMeasureMatrix (AgentGroup)`, using the given simulator `sim` for creating internal probes.

```
public GroupVolumeStatMeasureMatrix (AgentGroup group, int numTypes)
```

Constructs a new agent-group volume statistical probe observing the agent group `group`, and supporting `numTypes` contact types.

Parameters

`group` the observed agent group.

`numTypes` the number of contact types.

Throws

`IllegalArgumentException` if `numTypes` is negative.

```
public GroupVolumeStatMeasureMatrix (Simulator sim, AgentGroup group, int  
                                     numTypes)
```

Equivalent to `GroupVolumeStatMeasureMatrix (AgentGroup, int)`, using the given simulator `sim` for creating internal probes.

Methods

```
public void setNumMeasures (int nm)
```

Throws an `UnsupportedOperationException`.

Throws

`UnsupportedOperationException` if this method is called.

```
public void setNumPeriods (int np)
```

Throws an `UnsupportedOperationException`.

Throws

`UnsupportedOperationException` if this method is called.

```
public static MeasureSet getServiceVolumeMeasureSet (MeasureMatrix[] vcalc)
```

Returns a measure set regrouping the service volumes for several agent groups and computing the sum. Row `r` of the resulting matrix corresponds to the service volume stored in `vcalc[r]`, and the last row contains the sum of the service volumes.

Parameter

`vcalc` the agent group volume matrices.

Returns the service volume measure set.

```
public static MeasureSet getServiceVolumeMeasureSet (MeasureMatrix[] vcalc,
                                                    int numTypes)
```

Returns a measure set regrouping the service volumes stored in `vcalc` for `numTypes` contact types. Row `numTypes*i + k` of the resulting measure set corresponds to the integral of the number of busy agents in group `i` serving contacts of type `k`. If the measure set is computing the sum row (the default), row `numTypes*vcalc.length + k` gives the integral of the total number of agents serving contacts with type `k`.

Parameters

`vcalc` the agent group volume matrices.

`numTypes` the number of contact types.

Returns the service volume measure set.

```
public static MeasureSet getIdleVolumeMeasureSet (MeasureMatrix[] vcalc)
```

Returns a measure set regrouping the idle volumes for several agent groups and computing the sum. Row `r` of the resulting matrix corresponds to the idle volume stored in `vcalc[r]`, and the last row contains the sum of the idle volumes.

Parameter

`vcalc` the agent group volume matrices.

Returns the idle volume measure set.

```
public static MeasureSet getWorkingVolumeMeasureSet (MeasureMatrix[] vcalc)
```

Returns a measure set regrouping the working volumes for several agent groups. Row `r` of the resulting matrix corresponds to the working volume stored in `vcalc[r]`, and the last row contains the sum of the working volumes.

Parameter

`vcalc` the agent group volume matrices.

Returns the working volume measure set.

```
public static MeasureSet getWorkingVolumeMeasureSet (MeasureMatrix[] vcalc,
                                                    int numTypes)
```

Returns a measure set regrouping the working volumes stored in `vcalc`, with each working volume repeated `numTypes` times. This is used to create a measure set matching with `getServiceVolumeMeasureSet (MeasureMatrix[], int)` to compute per-contact type agent's occupancy ratios. If the measure set is computing the sum rows (the default), the last `numTypes` rows contain the sum of the working volumes for all agents.

Parameters

`vcalc` the agent group volume matrices.

`numTypes` the number of contact types.

Returns the working volume measure set.

```
public static MeasureSet getTotalVolumeMeasureSet (MeasureMatrix[] vcalc)
```

Returns a measure set regrouping the total volumes for several agent groups. Row **r** of the resulting matrix corresponds to the total volume stored in **vcalc[r]**, and the last row contains the sum of the total volumes.

Parameter

vcalc the agent group volume matrices.

Returns the total volume measure set.

```
public static MeasureSet getTotalVolumeMeasureSet (MeasureMatrix[] vcalc,  
                                                    int numTypes)
```

Returns a measure set regrouping the total volumes stored in **vcalc**, with each total volume repeated **numTypes** times. This is used to create a measure set matching with **getServiceVolumeMeasureSet (MeasureMatrix[], int)** to compute per-contact type agent's occupancy ratios. If the measure set is computing the sum rows (the default), the last **numTypes** rows contain the sum of the total volumes for all agents.

Parameters

vcalc the agent group volume matrices.

numTypes the number of contact types.

Returns the total volume measure set.

ContactTimeGenerator

Value generator for the communication times of contacts. This implementation simply calls the `Contact.getDefaultContactTime (int)` method to get the contact times. For each new agent group, such a value generator is created and used by default.

```
package umontreal.iro.lecuyer.contactcenters.server;

public class ContactTimeGenerator implements ValueGenerator
```

Constructors

```
public ContactTimeGenerator (AgentGroup group)
```

Constructs a contact time generator returning the same contact time for each contact type.

Parameter

`group` the associated agent group.

```
public ContactTimeGenerator (AgentGroup group, double[] mult)
```

Constructs a new contact time generator with a different multiplier for each contact type. When a contact time is required for a contact of type `k`, the result of `Contact.getDefaultContactTime()` is multiplied by `mult[k]`.

Parameters

`group` the associated agent group.

`mult` the vector contact time multipliers.

Methods

```
public AgentGroup getAgentGroup()
```

Returns the reference to the associated agent group.

Returns the associated agent group.

```
public void setAgentGroup (AgentGroup group)
```

Sets the associated agent group to `group`.

Parameter

`group` the new associated agent group.

```
public double[] getMultipliers()
```

Returns the vector of multipliers for this contact time generator. For contact type `k`, the multiplier of the contact times is given by the element with index `k` in the array. If this returns `null`, contact times all have multiplier 1.

Returns the vector of contact times multipliers.

```
public void setMultipliers (double[] mult)
```

Sets the contact time multiplier for each contact type to `mult`.

Parameter

`mult` the new vector of contact times multipliers.

AfterContactTimeGenerator

Value generator for the after-contact time of contacts. This implementation simply calls the `Contact.getDefaultAfterContactTime()` method to get the after contact times. For each new agent group, such a value generator is created and used by default.

```
package umontreal.iro.lecuyer.contactcenters.server;

public class AfterContactTimeGenerator implements ValueGenerator
```

Constructors

```
public AfterContactTimeGenerator (AgentGroup group)
```

Constructs an after-contact time generator returning the same after-contact time for each contact type.

Parameter

`group` the associated agent group.

```
public AfterContactTimeGenerator (AgentGroup group, double[] mult)
```

Constructs a new after-contact time generator with a different multiplier for each contact type. When an after-contact time is required for a contact of type `k`, the result of `Contact.getDefaultAfterContactTime()` is multiplied by `mult[k]`.

Parameters

`group` the associated agent group.

`mult` the vector of after-contact time multipliers.

Methods

```
public double[] getMultipliers()
```

Returns the vector of multipliers for this after-contact time generator. For contact type `k`, the multiplier of the after-contact times is given by the element with index `k` in the array. If this returns `null`, after-contact times all have multiplier 1.

Returns the vector of after-contact times multipliers.

```
public void setMultipliers (double[] mult)
```

Sets the after-contact time multiplier for each contact type to `mult`.

Parameter

`mult` the new vector of after-contact times multipliers.

```
public AgentGroup getAgentGroup()
```

Returns the reference to the associated agent group.

Returns the associated agent group.

```
public void setAgentGroup (AgentGroup group)
```

Sets the associated agent group to `group`.

Parameter

`group` the new associated agent group.

AgentGroupState

Represents the state of an agent group, i.e., the contacts being served at a specific simulation time.

```
package umontreal.iro.lecuyer.contactcenters.server;  
  
public class AgentGroupState
```

Constructor

```
protected AgentGroupState (AgentGroup group)
```

Constructs a new state object holding the state of the agent group `group`.

Parameter

`group` the agent group to be saved.

Methods

```
public EndServiceEvent[] getContactsInService()
```

Returns the end-service events representing the contacts being served at the time the state was saved.

Returns the array of end-service events representing contacts in service.

```
public double getEfficiency()
```

Returns the efficiency of the agent group at the time of state saving.

Returns the efficiency at the time the state was saved.

```
public int getNumAgents()
```

Returns the number of agents in the agent group at the time of state saving.

Returns the number of agents at the time the state was saved.

```
public int getNumFreeAgents()
```

Returns the number of free agents in the agent group at the time of state saving.

Returns the number of free agents at the time the state was saved.

```
public int getNumGhostAgents()
```

Returns the number of ghost agents in the agent group at the time of state saving.

Returns the number of ghost agents at the time the state was saved.

DetailedAgentGroupState

Represents the state of a detailed agent group.

```
package umontreal.iro.lecuyer.contactcenters.server;  
  
public class DetailedAgentGroupState extends AgentGroupState
```

Constructor

```
protected DetailedAgentGroupState (DetailedAgentGroup group)
```

Constructs a new agent group state object holding state information about the agent group group.

Parameter

group the agent group to save state.

Methods

```
public AgentState[] getBusyAgents()
```

Returns the state information for each busy agent in the group at the time of state saving.

Returns the state information about busy agents.

```
public AgentState[] getGhostAgents()
```

Returns the state information for the ghost agents in the group, at time of state saving.

Returns the state information about ghost agents.

```
public AgentState[] getIdleAgents()
```

Returns the state information about idle agents in the group, at the time of state saving.

Returns the state information about idle agents.

AgentState

Represents the state of an agent in a group.

```
package umontreal.iro.lecuyer.contactcenters.server;  
  
public class AgentState
```

Methods

```
public void restore()
```

Restores the state of the agent attached to this state object.

```
public Agent getAgent()
```

Returns the agent for which the state was saved.

Returns the agent for which the state was saved.

```
public boolean wasAvailable()
```

Determines the availability status of the agent at the time of state saving.

Returns `true` if the agent was available for serving contacts at the time of state saving, `false` otherwise.

```
public double getFirstLoginTime()
```

Returns the first login time of the agent at the time the state was saved.

Returns the first login time of the agent.

```
public double getIdleSimTime()
```

Returns the last simulation time the agent became idle, at the time of state saving.

Returns the last idle time of the agent.

StartServiceEvent

Represents an event that restarts the service of a contact. Service can be restarted in its communication phase, or in the after-contact work. This is used for state restoration of an agent group.

```
package umontreal.iro.lecuyer.contactcenters.server;
```

```
public class StartServiceEvent extends Event
```

Constructors

```
public StartServiceEvent (EndServiceEvent oldEndServiceEvent)
```

Constructs a new start-service event that will put the contact in service represented by `oldEndServiceEvent` in the target agent group given by `EndServiceEvent.getAgentGroup()`.

Parameter

`oldEndServiceEvent` the old end-service event.

```
public StartServiceEvent (AgentGroup targetGroup, EndServiceEvent  
                        oldEndServiceEvent)
```

Constructs a new start-service event that will put the contact in service represented by `oldEndServiceEvent` in the target agent group `targetGroup`.

Parameters

`targetGroup` the target agent group.

`oldEndServiceEvent` the old end-service event.

```
public StartServiceEvent (AgentGroup targetGroup, Contact contact, double  
                        contactTime, int ecType)
```

Constructs an event that will call `targetGroup.serve (contact, contactTime, ecType)` when it happens.

Parameters

`targetGroup` the target agent group.

`contact` the contact to serve.

`contactTime` the contact time.

`ecType` the end-contact type.

Throws

`NullPointerException` if `targetGroup` or `contact` are null.

`IllegalArgumentException` if `contactTime` is negative.

```
public StartServiceEvent (AgentGroup targetGroup, Contact contact, double
                           contactTime, int ecType, double afterContactTime,
                           int esType)
```

Constructs an event that will call `targetGroup.serve (contact, contactTime, ecType, afterContactTime, esType)` when it happens.

Parameters

`targetGroup` the target agent group.

`contact` the contact to serve.

`contactTime` the contact time.

`ecType` the end-contact type.

`afterContactTime` the after-contact time.

`esType` the end-service type.

Throws

`NullPointerException` if `targetGroup` or `contact` are null.

`IllegalArgumentException` if `contactTime` or `afterContactTime` are negative.

```
public StartServiceEvent (Agent targetAgent, Contact contact, double
                           contactTime, int ecType)
```

Constructs an event that will call `targetAgent.getGroup().serve (contact, targetAgent, contactTime, ecType)` when it happens.

Parameters

`targetAgent` the target agent.

`contact` the contact to serve.

`contactTime` the contact time.

`ecType` the end-contact type.

Throws

`NullPointerException` if `targetAgent` or `contact` are null.

`IllegalArgumentException` if `contactTime` is negative.

```
public StartServiceEvent (Agent targetAgent, Contact contact, double
                           contactTime, int ecType, double afterContactTime,
                           int esType)
```

Constructs an event that will call `targetAgent.getGroup().serve (contact, targetAgent, contactTime, ecType, afterContactTime, esType)` when it happens.

Parameters

`targetAgent` the target agent.

`contact` the contact to serve.

`contactTime` the contact time.

`ecType` the end-contact type.

`afterContactTime` the after-contact time.

`esType` the end-service type.

Throws

`NullPointerException` if `targetAgent` or `contact` are null.

`IllegalArgumentException` if `contactTime` or `afterContactTime` are negative.

Methods

```
public AgentGroup getTargetAgentGroup()
```

Returns the agent group that will receive the contact stored into the attached end-service event.

Returns the target agent group.

```
public Agent getTargetAgent()
```

Returns the target agent of this event, or null if no target agent was specified.

Returns the target agent.

```
public Contact getContact()
```

Returns the contact being served.

Returns the contact being served.

```
public double getScheduledContactTime()
```

Returns the scheduled duration of the communication between the contact and an agent.

Returns the scheduled contact time.

```
public double getScheduledAfterContactTime()
```

Returns the scheduled after-contact time. If the after-contact time was not set, an `IllegalStateException` is thrown.

Returns the scheduled after-contact time.

Throws

`IllegalStateException` if the after-contact time was not set.

```
public int getScheduledEndContactType()
```

Returns the type of contact termination that will occur when the end-service event happens for the first time.

Returns the scheduled end-contact type.

```
public int getScheduledEndServiceType()
```

Returns the type of the service termination that will occur when the end-service event happens for the second time.

Returns the scheduled end-service type.

```
public boolean contactDone()
```

Determines if the communication is finished between the contact and the agent.

Returns `true` if the contact was served, `false` otherwise.

```
public EndServiceEvent getNewEndServiceEvent()
```

Returns the end-service event representing the contact's restarted service. This returns a non-null value only after the execution of the `actions()` method.

Returns the new dequeue event.

SetNumAgentsEvent

Represents a simulation that sets the number of agents and agents' efficiency in an agent group.

```
package umontreal.iro.lecuyer.contactcenters.server;  
  
public class SetNumAgentsEvent extends Event
```

Constructors

```
public SetNumAgentsEvent (AgentGroup group, int numAgents, double  
                           efficiency)
```

Constructs a new set-num-agents event that sets the number of agents in the group `group` to `numAgents`, and the efficiency factor to `efficiency`.

Parameters

`group` the target agent group.

`numAgents` the number of agents in the group after the event occurs.

`efficiency` the efficiency after the event occurs.

```
public SetNumAgentsEvent (Simulator sim, AgentGroup group, int numAgents,  
                           double efficiency)
```

Equivalent to `SetNumAgentsEvent (AgentGroup, int, double)`, using the given simulator `sim`.

Methods

```
public AgentGroup getTargetAgentGroup()
```

Returns the agent group affected by this event.

Returns the target agent group.

```
public int getNumAgents()
```

Returns the number of agents in the target group after the event occurs.

Returns the desired number of agents in the target group.

```
public double getEfficiency()
```

Returns the agents' efficiency in the target group after this event occurs.

Returns the desired efficiency in the target agent group.

RestoreAgentsEvent

Represents an event that restores the state of busy and ghost agents after the service of contacts are started, during state restoration.

```
package umontreal.iro.lecuyer.contactcenters.server;  
  
public class RestoreAgentsEvent extends Event
```

Constructor

```
public RestoreAgentsEvent (DetailedAgentGroup dgroup, AgentState[]  
                           busyAgents, AgentState[] ghostAgents)
```

Constructs a new agent restoration event concerning agents in the group `dgroup`. When the event occurs, the state (available, last idle time, etc.) will be restored for all agents referred by `busyAgents` and `ghostAgents` while agents referred by `ghostAgents` will be removed from the agent group.

Parameters

`dgroup` the agent group affected by the restoration.

`busyAgents` the busy agents to be restored.

`ghostAgents` the ghost agents to be removed from the group.

Methods

```
public DetailedAgentGroup getTargetAgentGroup()
```

Returns the agent group affected by this event.

Returns the target agent group.

```
public AgentState[] getBusyAgents()
```

Returns the state of the busy agents that will be restored when the event occurs.

Returns the state of the busy agents.

```
public AgentState[] getGhostAgents()
```

Returns the state of the ghost agents that will be restored when this event occurs.

Returns the state of ghost agents.

Package `umontreal.iro.lecuyer.contactcenters.dialer`

Manages a predictive dialer capable of making outbound contacts. A *predictive dialer* is normally used to generate outbound calls. The dialer's policy determines the number of calls to try on each occasion, and supplies a list to extract them from. Each extracted contact is then tested for success or failure. The dialer defines separate lists of new-contact listeners for right party connects, and failed calls.

This package provides the `Dialer` representing the predictive dialer. Any dialing policy is an implementation of the interface `DialerPolicy` and dialer lists are represented by `DialerList` objects. The package provides implementations of dialer lists as well as commonly used dialing policies.

Dialer

Represents a predictive dialer making outbound contacts. A *predictive dialer* is normally used to generate outbound calls. The dialer's policy determines the number of calls to try on each occasion (as a function of the system's state), and supplies a list to extract them from. This list could be produced by a contact factory and is often assumed to be infinite for simplicity. Such lists could also be constructed from customer contacts who left a message, who were disconnected, etc.

For each call extracted from the dialer list, a success test is performed. This test succeeds with a probability being fixed or depending on the tested call, and the state of the system. Successful calls represent right party connects whereas failed calls represent wrong party connects and connection failures. The dialer generates a random delay representing the time between the beginning of dialing and the success or failure. This delay may depend on the success indicator, the call itself, the current time, etc. An event for broadcasting the call to registered listeners is then scheduled to occur at the time of success or failure.

The dialer defines separate lists of new-contact listeners for right party connects, and failed calls. Usually, only right party connects reach the router, but statistical collectors may need to listen to failed calls as well.

Note: the order in which `NewContactListener` implementations are notified is unspecified, and a new-contact listener modifying a list of listeners could result in unpredictable behavior.

```
package umontreal.iro.lecuyer.contactcenters.dialer;

public class Dialer implements ContactSource
```

Constructors

```
public Dialer (DialerPolicy policy, RandomStream streamReach,
               ValueGenerator probReach)
```

Constructs a new dialer using the dialer policy `policy`, the random stream `streamReach` to determine if a dialed call reaches the right party, and with 0 reach and fail times.

Parameters

`policy` the dialer's policy being used.

`streamReach` the random number stream used to determine the success of a dial.

`probReach` the probability of reaching the right party.

Throws

`NullPointerException` if any argument is null.

```
public Dialer (Simulator sim, DialerPolicy policy, RandomStream
               streamReach, ValueGenerator probReach)
```

Equivalent to `Dialer (DialerPolicy, RandomStream, ValueGenerator)`, with the given user-defined simulator `sim`.

Parameters

`sim` the simulator attached to the dialer.

`policy` the dialer's policy being used.

`streamReach` the random number stream used to determine the success of a dial.

`probReach` the probability of reaching the right party.

Throws

`NullPointerException` if any argument is null.

```
public Dialer (DialerPolicy policy, RandomStream streamReach,
               ValueGenerator probReach, ValueGenerator reachTimeGen,
               ValueGenerator failTimeGen)
```

Constructs a new dialer using the dialer policy `policy`. The random stream `streamReach` is used to determine if a call is reached, `reachTimeGen` and `failTimeGen` compute the reach and fail times, respectively, i.e., the simulation time between the call to `dial()` and the notification to the appropriate new-contact listeners.

Parameters

`policy` the dialer policy being used.

`streamReach` the random number stream used to compute the status of a dial.

`probReach` the probability of successful contact.

`reachTimeGen` the value generator for the time between dialing and reaching.

`failTimeGen` the value generator for the time between dialing and failing.

Throws

`NullPointerException` if any argument is null.

```
public Dialer (Simulator sim, DialerPolicy policy, RandomStream
               streamReach, ValueGenerator probReach, ValueGenerator
               reachTimeGen, ValueGenerator failTimeGen)
```

Equivalent to `Dialer (DialerPolicy, RandomStream, ValueGenerator, ValueGenerator, ValueGenerator)`, using the given simulator `sim`.

Parameters

`sim` the simulator attached to the dialer.

`policy` the dialer policy being used.

`streamReach` the random number stream used to compute the status of a dial.

`probReach` the probability of successful contact.

`reachTimeGen` the value generator for the time between dialing and reaching.

`failTimeGen` the value generator for the time between dialing and failing.

Throws

`NullPointerException` if any argument is `null`.

Methods

```
public void addNewContactListener (NewContactListener listener)
    Calls addReachListener (NewContactListener).
```

Parameter

`listener` the new-contact listener being added.

Throws

`NullPointerException` if `listener` is `null`.

```
public void removeNewContactListener (NewContactListener listener)
    Calls removeReachListener (NewContactListener).
```

Parameter

`listener` the new-contact listener being removed.

```
public void clearNewContactListeners()
    Calls clearReachListeners().
```

```
public List<NewContactListener> getNewContactListeners()
    Returns the result of getReachListeners().
```

```
public void addReachListener (NewContactListener listener)
    Adds the new-contact listener listener which will be notified upon right party connects.
```

Parameter

`listener` the new-contact listener being added.

Throws

`NullPointerException` if `listener` is null.

```
public void removeReachListener (NewContactListener listener)
```

Removes the new-contact listener `listener` from the list of listeners being notified upon right party connects.

Parameter

`listener` the new-contact listener being removed.

```
public void clearReachListeners()
```

Removes all new-contact listeners being notified when this dialer makes a right party connect.

```
public List<NewContactListener> getReachListeners()
```

Returns an unmodifiable list containing all the new-contact listeners notified when a right-party connect occurs.

Returns the list of all registered new-contact listeners.

```
public void addFailListener (NewContactListener listener)
```

Adds the new-contact listener `listener` which will be notified upon wrong party connects or connection failures.

Parameter

`listener` the new-contact listener being added.

Throws

`NullPointerException` if `listener` is null.

```
public void removeFailListener (NewContactListener listener)
```

Removes the new-contact listener `listener` from the list of listeners being notified upon wrong party connects or connection failures.

Parameter

`listener` the new-contact listener being removed.

```
public void clearFailListeners()
```

Removes all new-contact listeners being notified when this dialer fails to make a contact.

```
public List<NewContactListener> getFailListeners()
```

Returns an unmodifiable list containing all the new-contact listeners notified when the dialer fails making a contact.

Returns the list of all registered new-contact listeners.

```
public DialerPolicy getDialerPolicy()
```

Returns the dialing policy used by this dialer.

Returns the used dialer's policy.

```
public void setDialerPolicy (DialerPolicy policy)
```

Sets the dialing policy to `policy`.

Parameter

`policy` the new dialer's policy.

Throws

`NullPointerException` if `policy` is null.

```
public RandomStream getStreamReach()
```

Returns the random stream used to determine if a called person is reached or not.

Returns the stream used to determine if a called person is reached.

```
public void setStreamReach (RandomStream streamReach)
```

Sets the stream used to determine if a called person is reached to `streamReach`.

Parameter

`streamReach` the new stream for success tests.

Throws

`NullPointerException` if `streamReach` is null.

```
public ValueGenerator getProbReachGenerator()
```

Returns the value generator for the probability of a call to be successful, i.e., the probability of right party connect.

Returns the value generator for the reach probability.

```
public void setProbReachGenerator (ValueGenerator probReach)
```

Sets the value generator for right party connect probabilities to `probReach`.

Parameter

`probReach` the value generator for right party connect probabilities.

Throws

`NullPointerException` if `probReach` is null.

```
public ValueGenerator getReachTimeGenerator()
```

Returns the value generator for the reach times. A reach time corresponds to the simulation time from the call to `dial()` to the notification of the successful call to the listeners.

Returns the value generator for reach times.

```
public void setReachTimeGenerator (ValueGenerator reachTimeGen)
```

Sets the value generator for reach times to `reachTimeGen`. If `reachTimeGen` is `null`, the dial delay for successful calls is reset to 0.

Parameter

`reachTimeGen` the value generator for reach times.

```
public ValueGenerator getFailTimeGenerator()
```

Returns the value generator for the fail times. A fail time corresponds to the simulation time from the call to `dial()` to the notification of the failed call to the listeners.

Returns the value generator for fail times.

```
public void setFailTimeGenerator (ValueGenerator failTimeGen)
```

Sets the value generator for fail times to `failTimeGen`. If `reachTimeGen` is `null`, the dial delay for successful calls is reset to 0.

Parameter

`failTimeGen` the value generator for fail times.

```
public boolean isSuccessful (Contact contact)
```

Determines if the call represented by `contact` is a right party connect. Returns `true` if the call is successful, or `false` otherwise.

The default implementation uses the random stream returned by `getStreamReach()` to return `true` with some probability. The probability of right party connect is generated using the value generator returned by `getProbReachGenerator()`.

Parameter

`contact` the contact being tested.

Returns the success indicator.

```
public boolean isUsingNumActionsEvents()
```

Determines if the `dial()` method subtracts the number of action events returned by `getNumActionEvents()` from the return value of `DialerPolicy.getNumDials (Dialer)` in order to determine the number of calls to dial. When dial delays are large enough for the dialer to start often while phone numbers are being composed, the agents of the contact center might receive too many calls to serve, which results in a large number of mismatches. If this flag is enabled (the default), the dialer will take into account the number of calls for which dialing is in progress while determining the number of additional calls to dial.

Returns `true` if the number of action events must be taken into account while dialing.

```
public void setUsingNumActionEvents (boolean useNumActionEvents)
```

Sets the flag for taking the number of action events into account while dialing to `useNumActionEvents`.

Parameter

`useNumActionEvents` the new value of the flag.

See also `isUsingNumActionsEvents()`

public void dial()

Instructs the dialer to try performing outbound calls. This should be called at the end of a service, or at any time the number of agents capable of serving outbound calls increases. This method does nothing if the dialer is disabled.

The method uses the dialer's policy to get the appropriate number of calls to dial as well as the dialer list. The contact objects representing the calls being made are removed from the dialer list, and each call is tested using `isSuccessful (Contact)`. After the success indicator is determined, a corresponding dial delay is generated, and an event is scheduled to happen if the delay is non-zero. After the delay is elapsed, the appropriate new-contact listeners are notified about the new call.

protected void notifyListeners (Contact contact, boolean success)

Notifies registered new-contact listeners about the success or failure of the contact `contact`.

Parameters

`contact` the contact to broadcast.

`success` the success indicator.

public void stopDial()

Stops any ongoing dialing of calls. This can be called when the simulation program knows that if the called persons are reached, a mismatch will occur. Cancelled calls are notified as failed calls to the appropriate listeners if dialer action events are kept. However, if the dialer does not keep track of the action events, cancelled calls are lost without any notification.

public DialerState save()

Saves the state of this dialer and returns a state object containing the information.

Returns the state of the dialer.

public void restore (DialerState state)

Restores the state of this dialer with state information included in `state`.

Parameter

`state` the saved state of the dialer.

public boolean isKeepingActionEvents()

Determines if this dialer is keeping the action events. If this returns `true`, the `getActionEvents()` method can be used to return a set containing the events. Otherwise, the action events are stored in the event list only, and cannot be enumerated by the dialer. By default, the events are not stored by the dialer.

Returns the keep action events indicator.

```
public void setKeepingActionEvents (boolean keepActionEvents)
```

Sets the keep-dial-events indicator to `keepActionEvents`.

Parameter

`keepActionEvents` the new value of the indicator.

See also `isKeepingActionEvents()`

```
public Iterator<DialerActionEvent> dialerActionEventsIterator  
( )
```

Constructs and returns an iterator for the dialer-action events. If `isKeepingActionEvents()` returns `true`, the iterator is constructed from the set returned by `getActionEvents()`. Otherwise, an iterator traversing the event list and filtering the appropriate events is constructed and returned.

Returns the iterator for dialer-action events.

```
public Set<DialerActionEvent> getActionEvents()
```

Returns a set containing all the currently scheduled `DialerActionEvent` objects. If the dialer does not keep track of these events, an `IllegalStateException` is thrown.

Returns the set of dialer action events.

```
public int getNumActionEvents()
```

Returns the number of action events currently scheduled by this dialer. This corresponds to the number of calls the dialer is currently attempting.

Returns the current number of action events.

```
public void startNoDial()
```

This is the same as `start()`, except that no call to `dial()` is made after the dialer is started. `dial()` will then be called only when an agent becomes free.

DialerActionEvent

This event occurs when the dialer reached or failed to reach a called person. Such events are scheduled by the `Dialer.dial()` method if generated dial delays are greater than zero.

```
package umontreal.iro.lecuyer.contactcenters.dialer;
```

```
public class DialerActionEvent extends Event
    implements Cloneable
```

Constructor

```
public DialerActionEvent (Dialer dialer, Contact contact, boolean success)
```

Constructs a new dialer action event for contact `contact` with success indicator `success`. When the event occurs, if `success` is `true`, the contact is notified to new-contact listeners for right-party connect. When `success` is `false`, a failed contact is notified to the appropriate new-contact listeners.

Parameters

`dialer` the associated dialer.

`contact` the contact object representing the call being tried.

`success` the success indicator.

Methods

```
public Contact getContact()
```

Returns the contact object representing the called person.

Returns the concerned contact.

```
public Dialer getDialer()
```

Returns the dialer this event is attached to.

Returns the attached dialer.

```
public boolean isSuccessful()
```

Returns `true` if a right party connect will occur at the time of this event. Otherwise, returns `false`.

Returns the success indicator.

```
public boolean isObsolete()
```

Determines if this event is obsolete. When calling `Dialer.init()`, some action events might still be in the simulator's event list. One must use this method in `actions()` to test if this event is obsolete. If that returns `true`, one should return immediately.

Returns `true` for an obsolete event, `false` otherwise.

DialerPolicy

Represents a dialer's policy to determine the outbound calls to try on each occasion. A dialer's policy works as follows: each time the dialer is triggered, using the `Dialer.dial()` method, it uses the dialer policy to get the number of calls to try. It then uses the policy to obtain a dialer list from which to extract calls.

The simplest dialer policies compute and return a single number of calls to dial, e.g., by looking at the number of free outbound agents. A fixed dialer list is then returned to allow the dialer to get the contacts. However, most complex policies might generate a list of contacts each time the dialer is triggered.

```
package umontreal.iro.lecuyer.contactcenters.dialer;
```

```
public interface DialerPolicy
```

Methods

```
public int getNumDials (Dialer dialer)
```

Returns the number of calls the dialer should try to make simultaneously at the current simulation time.

If `Dialer.isUsingNumActionsEvents()` returns `true`, this method must take into account the current number of action events while determining the additional number of calls to dial. In the simplest and most common cases, the method subtracts the result of `Dialer.getNumActionEvents()` to the number of calls to dial. However, in some cases, it might be necessary to use `Dialer.getNumActionEvents (int)` to get the number of action events for each contact type individually.

Parameter

`dialer` the triggered dialer.

Returns the number of calls the dialer should try to make.

```
public DialerList getDialerList (Dialer dialer)
```

Returns the dialer list from which contacts have to be removed from, at the current simulation time. This list should not be stored into another object since it could be constructed dynamically when `getNumDials (Dialer)` is called.

Parameter

`dialer` the dialer for which the dialer list is required.

Returns the associated dialer list.

```
public void init (Dialer dialer)
```

Initializes this dialer's policy for a new simulation replication. This method can be used, for example, to clear data structures containing information about a preceding simulation. This method should also clear the associated dialer list when appropriate.

Parameter

`dialer` the dialer which initialized this policy.

`public void dialerStarted (Dialer dialer)`

This method is called when the dialer using this policy is started.

Parameter

`dialer` the started dialer.

`public void dialerStopped (Dialer dialer)`

This method is called when the dialer using this policy is stopped.

Parameter

`dialer` the stopped dialer.

ConstantDialerPolicy

Represents a dialer's policy which always tries to make the same number of calls on each trial.

```
package umontreal.iro.lecuyer.contactcenters.dialer;  
  
public class ConstantDialerPolicy implements DialerPolicy
```

Constructor

```
public ConstantDialerPolicy (DialerList list, int n)
```

Constructs a new dialer's policy with dialer list `list`, and `n` calls to make on each trial.

Parameters

`list` the dialer list to extract calls from.

`n` the number of calls to make on each occasion.

Throws

`NullPointerException` if `list` is null.

`IllegalArgumentException` if `n` is negative.

Methods

```
public void setDialerList (DialerList list)
```

Sets the dialer list to `list`.

Parameter

`list` the new dialer list.

Throws

`NullPointerException` if `list` is null.

```
public void setNumDials (int n)
```

Sets the number of dialed contacts to `n`.

Parameter

`n` the number of calls to make upon each trial.

Throws

`IllegalArgumentException` if `n` is negative.

ThresholdDialerPolicy

Represents a threshold-based dialing policy selecting the number of calls to try based on the number of free agents in certain groups. Before trying to make calls, the policy determines the total number of free agents $N_F^T(t)$ in a *test set* of agent groups. If the number of free agents is greater than or equal to s_t , the policy counts the total number $N_F^D(t)$ of free agents in a *target set* of agent groups which may differ from the test set. If $N_F^D(t) \geq s_d$, the dialer tries to make $\max\{\text{Math.round}(\kappa N_F^D(t)) + c - a, 0\}$ calls, where $\kappa \in \mathbb{R}$ and $c \in \mathbb{N}$ are predefined numbers. The constant a is the result of `Dialer.getNumActionEvents()` if `Dialer.isUsingNumActionsEvents()` returns `true`, or 0 otherwise. Any parameter used by this policy can be changed at any time during the simulation.

```
package umontreal.iro.lecuyer.contactcenters.dialer;

public class ThresholdDialerPolicy implements DialerPolicy
```

Constructors

```
public ThresholdDialerPolicy (DialerList list, AgentGroupSet testGroups,
                             AgentGroupSet targetGroups, int minFreeTest,
                             int minFreeTarget, double kappa, int c)
```

Constructs a new dialer's policy with dialer list `list`, test set `testGroups`, and target set `targetGroups`. The free agents threshold for the test set is set to `minFreeTest`, the threshold is set to `minFreeTarget` for the target set, the multiplicative constant is set to `kappa`, and the additive constant to `c`.

Parameters

`list` the dialer list being used.

`testGroups` the test set of agent groups.

`targetGroups` the target set of agent groups.

`minFreeTest` the (inclusive) minimum number of free agents in the test set.

`minFreeTarget` the (inclusive) minimum number of free agents in the target set.

`kappa` the multiplicative constant.

`c` the additive constant.

Throws

NullPointerException if `list`, `testGroups`, or `targetGroups` are null.

IllegalArgumentException if the free agents threshold is negative.

```
public ThresholdDialerPolicy (DialerList list, AgentGroupSet testGroups,  
                             AgentGroupSet targetGroups, int minFreeTest,  
                             double kappa, int c)
```

Equivalent to `ThresholdDialerPolicy (list, testGroups, targetGroups, minFreeTest, 1, kappa, c)`.

Methods

```
public void setDialerList (DialerList list)
```

Sets the currently used dialer list to `list`.

Parameter

`list` the new dialer list.

Throws

NullPointerException if `list` is null.

```
public int getMinFreeAgentsTest()
```

Returns the minimal number of free agents s_t in the test set to try outbound calls.

Returns the minimal number of free agents in the test set to dial.

```
public void setMinFreeAgentsTest (int minFreeTest)
```

Sets the minimal number of free agents in the test set to `minFreeTest`.

Parameter

`minFreeTest` the new minimal number of free agents in the test set.

Throws

IllegalArgumentException if `minFreeTest` is negative.

```
public int getMinFreeAgentsTarget()
```

Returns the minimal number of free agents s_d in the target set to try outbound calls.

Returns the minimal number of free agents in the target set to dial.

```
public void setMinFreeAgentsTarget (int minFreeTarget)
```

Sets the minimal number of free agents in the target set to `minFreeTarget`.

Parameter

`minFreeTarget` the new minimal number of free agents in the target set.

Throws

`IllegalArgumentException` if `minFreeTarget` is negative.

`public double getKappa()`

Returns the current value of the multiplicative constant κ for this policy.

Returns the multiplicative constant.

`public void setKappa (double kappa)`

Sets the multiplicative constant κ to `kappa` for this dialer policy.

Parameter

`kappa` the new multiplicative constant.

`public int getC()`

Returns the current value of the additive constant c for this policy.

Returns the additive constant.

`public void setC (int c)`

Sets the additive constant c to `c` for this dialer's policy.

Parameter

`c` the new additive constant.

`public AgentGroupSet getTestSet()`

Returns the current test set of agent groups.

Returns the test set of agent groups.

`public void setTestSet (AgentGroupSet testGroups)`

Sets the test set of agent groups to `testGroups`.

Parameter

`testGroups` the new test set of agent groups.

Throws

`NullPointerException` if `testGroups` is null.

`public AgentGroupSet getTargetSet()`

Returns the current target set of agent groups.

Returns the target set of agent groups.

`public void setTargetSet (AgentGroupSet targetGroups)`

Sets the target set of agent groups to `targetGroups`.

Parameter

`targetGroups` the new target set of agent groups.

Throws

`NullPointerException` if `targetGroups` is null.

BadContactMismatchRatesDialerPolicy

Represents a threshold-based dialer's policy taking bad contact and mismatch rates into account for dialing, as used in Deslaurier's blend call center model [7]. This dialer's policy needs to be informed about the contact center's activity through two methods: `notifyInboundContact (Contact, boolean)`, and `notifyOutboundContact (Contact, boolean)`. When an inbound contact is processed (served, abandoned, or blocked) by the contact center, the `notifyInboundContact (Contact, boolean)` method needs to be called. When an outbound contact is processed, the `notifyOutboundContact (Contact, boolean)` method must be called. For both contact types, the user has to indicate this dialer's policy if the processed contact must be considered as good or bad. Usually, a *good* inbound contact is a contact meeting some service level requirements, e.g., having waited less than a certain time in queue. An outbound contact can be considered as good if it is not a mismatch.

When the dialer's policy is asked a number of calls to make, it gets the total number of free agents $N_F^T(t)$ in a *test set*. If this number is smaller than a given minimum s_t , no call is made. Otherwise, the dialer looks at the rate of bad inbound contacts in the last p periods of duration d . If this rate is smaller than or equal to a threshold s_i , the dialer's policy evaluates $N_F^D(t)$, the number of free agents in a *target set*. If this number is smaller than s_d , no call is made. Otherwise, the base number of calls to dial $n = \max\{\text{Math.round}(\kappa N_F^D(t)) + c - a, 0\}$ is computed. Then, if the rate of bad outbound contacts in the p last periods of duration d is smaller than or equal to a threshold s_o , $2n$ calls are made. Otherwise, n calls are made.

```
package umontreal.iro.lecuyer.contactcenters.dialer;
```

```
public class BadContactMismatchRatesDialerPolicy extends
    ThresholdDialerPolicy
```

Constructors

```
public BadContactMismatchRatesDialerPolicy (DialerList list, AgentGroupSet
                                             testSet, AgentGroupSet
                                             targetSet, int minFreeTest,
                                             int minFreeTarget, double
                                             maxBadContactRate, double
                                             mismatchRateThresh, int
                                             numCheckedPeriods, double
                                             checkedPeriodDuration)
```

This is the same as the constructor `BadContactMismatchRatesDialerPolicy (DialerList, AgentGroupSet, AgentGroupSet, int, int, double, int, double, double, int, double)`, with $\kappa = 1$ and $c = 0$.

Parameters

`list` the dialer list from which to get contacts.

`testSet` the test agent group set.

`targetSet` the target agent group set.

`minFreeTest` the minimal number of free agents in the test set.

`minFreeTarget` the minimal number of free agents in the target set.

`maxBadContactRate` the maximal rate of bad contacts.

`mismatchRateThresh` the mismatch rate threshold.

`numCheckedPeriods` the number of checked periods p .

`checkedPeriodDuration` the duration d of checked periods.

Throws

`NullPointerException` if `list`, `testSet` or `targetSet` are null.

`IllegalArgumentException` if a threshold on the number of agents, the number of checked periods or the duration of checked periods are negative, or a rate is negative or greater than 1.

```
public BadContactMismatchRatesDialerPolicy (DialerList list, AgentGroupSet
                                             testSet, AgentGroupSet
                                             targetSet, int minFreeTest,
                                             int minFreeTarget, double
                                             kappa, int c, double
                                             maxBadContactRate, double
                                             mismatchRateThresh, int
                                             numCheckedPeriods, double
                                             checkedPeriodDuration)
```

Constructs a new bad contact/mismatch rates dialer's policy with the dialer list `list`, test set `testSet`, target set `targetSet`. The minimal number of free agents is set to `minFreeTest` for the test set, and `minFreeTarget` for the target set. The maximal rate of bad contacts is set to `maxBadContactRate`, while the threshold for mismatch rate is `mismatchRateThresh`. To take its decisions, the policy uses rates for the last `numCheckedPeriods` periods of duration `checkedPeriodDuration`.

Parameters

`list` the dialer list from which to get contacts.

`testSet` the test agent group set.

`targetSet` the target agent group set.

`minFreeTest` the minimal number of free agents in the test set.

`minFreeTarget` the minimal number of free agents in the target set.

`kappa` the κ multiplicative constant.

`c` the c additive constant.

`maxBadContactRate` the maximal rate of bad contacts.

`mismatchRateThresh` the mismatch rate threshold.

`numCheckedPeriods` the number of checked periods p .

`checkedPeriodDuration` the duration d of checked periods.

Throws

`NullPointerException` if `list`, `testSet` or `targetSet` are null.

`IllegalArgumentException` if a threshold on the number of agents, the number of checked periods or the duration of checked periods are negative, or a rate is negative or greater than 1.

Methods

```
public double getMaxBadContactRate()
```

Returns the maximal rate of bad contacts s_i for this dialer's policy.

Returns the maximal rate of bad contacts.

```
public void setMaxBadContactRate (double maxBadContactRate)
```

Sets the maximal rate of bad contacts for this dialer's policy to `maxBadContactRate`.

Parameter

`maxBadContactRate` the new rate of bad contacts.

Throws

`IllegalArgumentException` if `maxBadContactRate` is smaller than 0 or greater than 1.

```
public double getMismatchRateThresh()
```

Returns the threshold on the mismatch rate s_o for this dialer's policy.

Returns the mismatch rate threshold.

```
public void setMismatchRateThresh (double mismatchRateThresh)
```

Sets the threshold on the mismatch rate to `mismatchRateThresh`.

Parameter

`mismatchRateThresh` the threshold on the mismatch rate.

Throws

`IllegalArgumentException` if `mismatchRateThresh` is smaller than 0 or greater than 1.

```
public int getNumCheckedPeriods()
```

Returns the number of checked periods p for this dialer's policy.

Returns the number of checked periods.

```
public void setNumCheckedPeriods (int numCheckedPeriods)
```

Sets the number of checked periods to `numCheckedPeriods`.

Parameter

`numCheckedPeriods` the number of checked periods.

Throws

`IllegalArgumentException` if `numCheckedPeriods` is negative or 0.

```
public double getCheckedPeriodDuration()
```

Returns the duration d of the checked periods.

Returns the checked period duration.

```
public void setCheckedPeriodDuration (double checkedPeriodDuration)
```

Sets the duration of the checked periods to `checkedPeriodDuration`.

Parameter

`checkedPeriodDuration` the duration of the checked periods.

Throws

`IllegalArgumentException` if `checkedPeriodDuration` is negative or 0.

```
public double getCurrentBadContactRate()
```

Gets the current bad contact rate as used by `getNumDials` (Dialer).

Returns the current bad contact rate.

```
public double getCurrentMismatchRate()
```

Returns the current mismatch rate as used by `getNumDials` (Dialer).

Returns the current mismatch rate.

```
public void notifyInboundContact (Contact contact, boolean bad)
```

Notify a processed inbound contact to this dialer's policy. The `bad` indicator determines if a bad contact is notified. The simulator must determine which contacts to notify as well as a definition of bad contacts. Usually, all inbound contacts are notified to the dialer, and bad contacts have a waiting time greater than some acceptable waiting time.

Parameters

`contact` the notified contact.

`bad` `true` if a bad contact is notified, `false` if a good contact is notified.

```
public void notifyOutboundContact (Contact contact, boolean m)
```

Notifies an outbound contact to this dialer policy. If `m` is `true`, the contact is a mismatch, i.e., it has balked (most often) or needed to wait before abandoning or being served. The application needs to decide which outbound contacts are notified and determine if contacts are mismatches or not. Usually, only outbound contacts produced by the dialer using this policy are notified, including right and wrong party connects.

Parameters

`contact` the notified contact.

`m` `true` if the notified contact is a mismatch, `false` otherwise.

```
public SumMatrixSW getInBadContactsSumMatrix()
```

Returns the matrix of sums counting the number of bad inbound contacts notified to this dialing policy.

Returns the matrix of sums for bad contacts.

```
public SumMatrixSW getInTotalSumMatrix()
```

Returns the matrix of sums counting the total number of inbound contacts notified to this dialing policy.

Returns the matrix of sums for the total number of contacts.

```
public SumMatrixSW getMismatchSumMatrix()
```

Returns the matrix of sums counting the number of mismatches notified to this dialing policy.

Returns the matrix of sums for the number of mismatches.

```
public SumMatrixSW getOutTotalSumMatrix()
```

Returns the matrix of sums counting the total number of outbound contacts for this dialing policy.

Returns the matrix of sums for the number of outbound contacts.

AgentsMoveDialerPolicy

Represents a dialer policy that dynamically moves agents from inbound to outbound groups to balance performance. This policy is inspired from a real dialer called SER's SmartAgent Manager. This dialer manages a subset of the I agent groups of the contact center by separating them into two categories: inbound agent groups and outbound agent groups. The inbound groups are assumed to serve inbound contacts only while the outbound groups process outbound contacts only. An inbound agent is an agent belonging to an inbound group while any outbound agent belongs to an outbound group. Consequently, an inbound agent is made outbound by removing it from its original inbound group, and adding it into an outbound group. A similar process is used to turn an outbound agent into an inbound one. This dialer policy performs such transfers in order to balance performance.

Note that this dialer policy does not impose outbound contacts to be routed to outbound agents, and inbound contacts to be sent to inbound agents. The routing policy must be configured separately to be consistent with the inbound and outbound agent groups managed by the dialer.

This policy required two different aspects to be specified: how contacts are dialed, and how agents are moved across groups. We will now describe these two aspects in more details.

The dialing process. Two algorithms are available for dialing: one simple method using no routing information, and one more elaborate method using the information. With the first and fastest method, the policy does not control the distribution of the dialed calls, which can result in many mismatches if agents can only serve a restricted subset of the calls. With the second method, the number of dialed calls of each type depends on the agents available to serve them. Both methods use a dialer list L to obtain calls to dial.

The first method works as follows. When the dialer is triggered, i.e., when it is requested to dial numbers, this policy computes the number of outbound agents managed by the dialer given by

$$N = \sum_{i=0}^{I-1} N_{F,i}(t) \mathbb{I}\{i \text{ is an outbound agent group managed by the dialer}\}.$$

The number of calls to dial is then obtained using $n = \text{round}(\kappa N) + c - a$ where $\kappa \in \mathbb{R}$, $c \in \mathbb{N}$, and $\text{round}(\cdot)$ rounds its argument to the nearest integer. The dialer schedules an *action event* for each call waiting a dial delay. If the number of action events is taken into account (the default), the constant a is the number of action events currently scheduled by the dialer. Otherwise, $a = 0$. An action event occurs when a call made by the dialer reaches a person or fails. The n calls to be dialed are extracted from the dialer list L .

The dialing method using routing information works as follows. For each managed agent group, the dialer determines a number of calls to dial using the number of free agents. It then sums up the number of calls m_k for each type, and constructs a dialer list L_2 containing at most m_k calls of type k . The calls are extracted from the dialer list L . The contents of the dialer list might be affected by limits imposed on the number of calls of each type.

The values of m_k are computed as follows. First, $m_k = 0$ for each value of k . Then, for each managed outbound agent group i , the dialer obtains $n_i = \text{round}(\kappa N_{F,i}(t)) + c$. Let K_i be the number of different types of calls agents in group i can serve, and $b_{k,i} = 1$ if and only if agents in group i can serve calls of type k . If $K_i > 0$, for each value of k , the value $\text{round}(b_{k,i}n_i/K_i)$ is added to m_k . Usually, $K_i = 1$ with this model, i.e., agents in each group i can serve a single outbound call type.

After each agent group is processed, if the dialer takes the number of action events into account, the number of action events concerning calls of type k is subtracted to each value of m_k , for each call type k .

Management of agents. This dialer regroups agent groups into virtual groups containing inbound and outbound agent groups. Let J be the total number of virtual agent groups, and $V_j(t) = I_j(t) + O_j(t)$ the number of agents in virtual group j at simulation time t , where $I_j(t)$ is the total number of inbound agents in virtual group j , and $O_j(t)$ is the total number of outbound agents in virtual group j at time t . This dialer policy never changes the virtual group of an agent; it only transfers agents to groups within the same virtual group.

Note that an agent group can only be in a single virtual group, for a single dialer.

Any external change to an agent group managed by this dialer policy is handled the same way as if the dialer never transferred agents from groups to groups. This requires the dialer to keep track of the number of agents transferred into or out of each managed group. The changes are performed as follows. If the number of agents is increased, agents are added to the concerned group. However, if the number of agents is reduced, the dialer first removes outbound agents, then inbound agents if the affected virtual group does not contain any more outbound agents. The order in which the agent groups are selected to remove agents from is random to avoid an agent group having priority over other groups. The only constraint on the order is the priority of outbound agents over inbound agents. When the dialer is stopped, every outbound agent becomes inbound, but busy outbound agents terminate their on-going services before they become inbound.

Two flags are available for this dialer policy: inbound-to-outbound flag, and outbound-to-inbound flag. These flags trigger procedures that can be considered as background processes, although they are implemented with simulation events. When the inbound-to-outbound flag is turned ON, the policy starts the following procedure for each virtual agent group j , each time the dialer is required to take a decision.

1. If the procedure is already running for virtual group j , stop.
2. Let τ be the delay between the last time an inbound agent in virtual group j became outbound, and the current simulation time. If $\tau < D_{OO,j}$, wait for $D_{OO,j} - \tau$.
3. Generate a random permutation of the inbound agent groups in the virtual group j .
4. For each inbound agent group i in the virtual group j , do the following. Agent groups are processed in the order given by the random permutation of the previous step. While $N_{I,i}(t) > 0$,

- (a) Select an agent A in group i with the following characteristics:
 - The agent is in group i (idle or busy) for a minimal time $D_{IO,j}$,
 - The idle time of the agent is greater than or equal to t_j ,
 - The number of idle inbound agents in virtual group j is greater than or equal to m_j ,
 - The number of outbound idle agents in virtual group j is smaller than M_j .
- (b) If no agent was selected at previous step, skip to next agent group.
- (c) Remove agent A from group i , select outbound agent group o with probability $p_{j,o}$, and add the agent A to group o .
- (d) Wait for a delay $D_{OO,j}$.

When the flag is turned OFF, every process moving inbound agents to outbound groups is stopped.

On the other hand, when the outbound-to-inbound flag is turned ON, the policy starts the following procedure for each virtual agent group j , each time the dialer is required to take a decision.

1. If the procedure is already running for group j , stop.
2. Let τ be delay between the last time an outbound agent in virtual group j became inbound, and the current simulation time. If $\tau < D_{II,j}$, wait for $D_{II,j} - \tau$.
3. Generate a random permutation of outbound agent groups.
4. For each outbound agent group o in virtual group j , do the following. Agent groups are processed in the order given by the random permutation generated at the previous step. While $N_{I,o}(t) > 0$,
 - (a) Select an agent A in group o with the following characteristic:
 - The agent is in group o (idle or busy) for a minimal time $D_{OI,j}$.
 - (b) If no agent was selected at previous step, skip to next agent group.
 - (c) Remove agent A from group o , select inbound agent group with probability $p_{j,i}$, and add agent A to group i .
 - (d) Wait for a delay $D_{II,j}$.

When the flag is turned OFF, every process moving outbound agents to inbound groups is stopped.

```
package umontreal.iro.lecuyer.contactcenters.dialer;
```

```
public class AgentsMoveDialerPolicy implements DialerPolicy
```

Constructor

```
public AgentsMoveDialerPolicy (DialerList list, AgentGroupInfo[] groupInfo,  
                               double kappa, int c)
```

Constructs a new dialer policy using the dialer list `list`, and agent group information `groupInfo`. Each `AgentGroupInfo` object results in a virtual group of agents for the dialer.

Parameters

`list` the dialer list.

`groupInfo` the agent group information.

Methods

```
public AgentGroupInfo[] getAgentGroupInfo()
```

Returns an array containing the references to the virtual agent groups managed by this dialer policy.

Returns the virtual agent groups for this policy.

```
public void dialerStopped (Dialer dialer)
```

Calls `init (Dialer)`.

```
public DialerList getDialerList (Dialer dialer)
```

Returns the dialer list associated with this policy.

```
public void setDialerList (DialerList list)
```

Sets the dialer list of this policy to `list`.

Parameter

`list` the new dialer list.

```
public int getNumDials (Dialer dialer)
```

This method returns the number of free agents in all outbound groups connected to this dialer.

```
public void init (Dialer dialer)
```

Makes every agent inbound when the dialer stops.

```
public boolean isInboundToOutboundStarted()
```

Determines if the inbound-to-outbound flag is turned ON.

Returns the status of the inbound-to-outbound flag.

```
public void startInboundToOutbound()
```

Turns the inbound-to-outbound flag on.

```
public void stopInboundToOutbound()
```

Turns the inbound-to-outbound flag off.

```
public boolean isOutboundToInboundStarted()
```

Determines if the outbound-to-inbound flag is turned ON.

Returns the status of the outbound-to-inbound flag.

```
public void startOutboundToInbound()
```

Turns the outbound-to-inbound flag on.

```
public void stopOutboundToInbound()
```

Turns the outbound-to-inbound flag off.

Nested class

```
public static class AgentGroupInfo
```

Represents a virtual agent group j for the `AgentsMoveDialerPolicy`. This class encapsulates information about inbound and outbound groups in the virtual group as well as thresholds, probabilities, and delays. It also implements methods to transfer agents from groups to groups.

Constructor

```
public AgentGroupInfo (AgentGroup[] inboundGroups, double[]
                        inboundGroupProbs, AgentGroup[] outboundGroups,
                        double[] outboundGroupProbs, RandomStream stream)
```

Constructs a new virtual agent group containing all inbound agent groups in `inboundGroups`, and all outbound agent groups in `outboundGroups`. The arrays `inboundGroupProbs` and `outboundGroupProbs` contain probabilities $p_{j,i}$ of selection of agent groups as targets for transfers. The random stream `stream` is used to generate random numbers for permutations, and for selecting target agent groups during transfers.

Parameters

inboundGroups the inbound agent group.

inboundGroupProbs the probabilities of selection for each inbound agent group when performing transfers.

outboundGroups the outbound agent group.

outboundGroupProbs the probabilities of selection for each outbound agent group when performing transfers.

Methods

```
public double getDelayInIn()
```

Returns the value of $D_{II,j}$, which defaults to 0.

```
public void setDelayInIn (double delayInIn)
```

Sets the value of $D_{II,j}$ to **delayInIn**.

```
public double getDelayInOut()
```

Returns the value of $D_{IO,j}$, which defaults to 0.

```
public void setDelayInOut (double delayInOut)
```

Sets the value of $D_{IO,j}$ to **delayInOut**.

```
public double getDelayOutIn()
```

Returns the value of $D_{OI,j}$, which defaults to 0.

```
public void setDelayOutIn (double delayOutIn)
```

Sets the value of $D_{OI,j}$ to **delayOutIn**.

```
public double getDelayOutOut()
```

Returns the value of $D_{OO,j}$, which defaults to 0.

```
public void setDelayOutOut (double delayOutOut)
```

Sets the value of $D_{OO,j}$ to **delayOutOut**.

```
public AgentGroup[] getInboundGroups()
```

Returns the inbound agent group associated with this information object.

```
public AgentGroup[] getOutboundGroups()
```

Returns the outbound agent group associated with this information object.

```
public int getMaximumIdleOutboundAgents()
```

Returns the value of M_j , which defaults to 0.

```
public void setMaximumIdleOutboundAgents (int maximumIdleAgents)
```

Sets the value of M_j to `maximumIdleAgents`.

```
public int getMinimumIdleInboundAgents()
```

Returns the value of m_j , which defaults to 0.

```
public void setMinimumIdleInboundAgents (int minimumIdleAgents)
```

Sets the value of m_j to `minimumIdleAgents`.

```
public double getMinimumIdleTime()
```

Returns the value of t_j , which defaults to 0.

```
public void setMinimumIdleTime (double minimumIdleTime)
```

Sets the value of t_j to `minimumIdleTime`.

```
public double[] getInboundGroupProbs()
```

Returns the probabilities $p_{j,i}$ of selection for each inbound agent group. Element k of the returned array corresponds to the probability associated with agent group k in the array returned by `getInboundGroups()`.

Returns the probability of selection for inbound agent groups.

```
public double[] getOutboundGroupProbs()
```

Returns the probabilities $p_{j,i}$ of selection for each outbound agent group. Element k of the returned array corresponds to the probability associated with agent group k in the array returned by `getOutboundGroups()`.

Returns the probability of selection for outbound agent groups.

```
public void transferToInbound (int n)
```

Transfers n agents from the outbound groups of this object to its inbound group. For each transfer, the order of outbound agent groups is chosen randomly to avoid an outbound group having priority over the others.

Parameter

n the number of agents to transfer.

```
public void transferToOutbound (int n)
```

Transfers n agents from the inbound groups of this object to its outbound group. For each transfer, the order of inbound agent groups is chosen randomly to avoid an inbound group having priority over the others.

Parameter

n the number of agents to transfer.

```
public boolean transferToInbound (int idxOut, Agent agent)
```

Transfers the agent **agent** to a randomly-chosen inbound agent group.

Parameter

agent the agent to transfer.

```
public boolean transferToOutbound (int idxIn, Agent agent)
```

Transfers the agent **agent** to a randomly-chosen outbound agent group.

Parameter

agent the agent to transfer.

```
public void startInboundToOutbound()
```

Starts the process moving inbound agents to outbound for the agent groups associated with this object. This method does nothing if the moving process is already started.

```
public void stopInboundToOutbound()
```

Stops the process moving inbound agents to the outbound group.

```
public void startOutboundToInbound()
```

Similar to `startInboundToOutbound()`, for the outbound-to-inbound process.

```
public void stopOutboundToInbound()
```

Similar to `stopInboundToOutbound()`, for the outbound-to-inbound process.

```
public void makeAllInbound()
```

Moves all outbound agents to the inbound group. Any busy outbound agent is marked to be moved after its on-going service if finished.

```
public void init()
```

Initializes both agent groups, and resets the fields storing the last time moves happened.

DialerList

Represents a list that contains and manages contacts to be made later by a dialer. This interface specifies methods to clear the current list, to remove the first contact of the list, and to compute the current size of the list. The contents of the dialer list depends on the implementation. For example, `InfiniteDialerList` always has an infinite size, and uses a user-supplied contact factory to obtain contacts. The `ContactListenerDialerList`, on the other hand, is backed by a fixed list which is populated by a contact listener. The contents of the dialer list could also change with time. For example, during some time intervals of the day, a limit on the maximal number of dialed contacts might be imposed by restricting the size of the dialer list.

Some dialer lists might also allow one to restrict the types of contacts that can be extracted from the list. This can be useful for some dialing policies managing contacts of several types, and composing specific numbers of contacts of each type.

A dialer list also implements the `ContactFactory` interface; the implementation of the `ContactFactory.newInstance()` method usually forwards the call to `removeFirst(int[])`. However, while the contact factory always creates new objects, a dialer list may extract them from a list with possibly finite size.

```
package umontreal.iro.lecuyer.contactcenters.dialer;

public interface DialerList extends ContactFactory
```

Methods

```
public int size (int[] contactTypes)
```

Returns the number of contacts of desired types stored into this dialer list. This method counts and returns the number of stored contacts whose type identifiers correspond to one of the elements in the given `contactTypes` array. If the array is `null`, the check is applied for all contact types. If the size of the list is infinite, this must return `Integer.MAX_VALUE`. If the dialer list does not allow restriction to specific contact types, this method throws an `UnsupportedOperationException`.

Parameter

`contactTypes` the array of desired contact types.

Returns the number of contacts in the dialer list.

Throws

UnsupportedOperationException if **contactTypes** is non-null while the dialer list does not support restrictions to specific contact types.

```
public void clear()
```

Clears the contents of this dialer list. This method does not always reset the size of the list to 0. For example, this method has no effect in the case of infinite dialer lists. For dialer lists with limits on the number of dialed contacts, this resets the size to the maximum number of contacts allowed.

```
public Contact removeFirst (int[] contactTypes)
```

Removes and returns the first contact with one of the desired types from the dialer list. If the list is empty or does not contain any contact of the desired types, this method must throw a **NoSuchElementException**. If **contactTypes** is **null**, any contact type is allowed. If **contactTypes** is non-null while the dialer list does not support restrictions to specific contact types, this throws an **UnsupportedOperationException**.

Parameter

contactTypes the array of desired contact types.

Returns the removed contact.

Throws

NoSuchElementException if the dialer list is empty.

UnsupportedOperationException if **contactTypes** is non-null while the dialer list does not support restrictions to specific contact types.

InfiniteDialerList

Implements the `DialerList` interface for an infinite dialer list whose elements are produced using a contact factory. This list can be used when there is no defined model for the calls made by the dialer.

```
package umontreal.iro.lecuyer.contactcenters.dialer;  
  
public class InfiniteDialerList implements DialerList
```

Constructor

```
public InfiniteDialerList (ContactFactory factory)
```

Constructs a new infinite dialer list whose contacts are instantiated using the contact factory `factory`.

Parameter

`factory` the contact factory used to instantiate contacts.

Throws

`NullPointerException` if `factory` is null.

Methods

```
public ContactFactory getContactFactory()
```

Returns the contact factory associated with this dialer list.

Returns the associated contact factory.

```
public void setContactFactory (ContactFactory factory)
```

Sets the contact factory used to instantiate contacts to `factory`.

Parameter

`factory` the new contact factory.

Throws

`NullPointerException` if `factory` is null.

ContactListenerDialerList

Implements the `DialerList` interface for a finite dialer list whose elements are obtained from an external source. Since this class implements the `NewContactListener` interface, it can be bound to an arrival process or any source of contacts. When a new contact is notified to this dialer list, it is added at the end of an internal ordered list for later use. When calling `removeFirst (int[])`, the first element from this internal list is returned.

```
package umontreal.iro.lecuyer.contactcenters.dialer;  
  
public class ContactListenerDialerList implements DialerList,  
        NewContactListener
```

Constructors

```
public ContactListenerDialerList()
```

Constructs a new empty dialer list implemented by a doubly-linked list. The used implementation is provided by the standard `LinkedList` class.

```
public ContactListenerDialerList (List<Contact> dialerList)
```

Constructs a new dialer list using the given `dialerList` to store the contacts. The given list should be empty or contain only `Contact` instances.

Parameter

`dialerList` the list used to store the contacts.

Throws

`NullPointerException` if `dialerList` is null.

Methods

```
public List<Contact> getList()
```

Returns the internal list containing the contacts to dial. This list should contain only non-null `Contact` instances.

Returns the internal dialer list.

```
public void setList (List<Contact> dialerList)
```

Sets the internal list of contacts to dial to `dialerList`.

Parameter

`dialerList` the list used to store the contacts.

Throws

NullPointerException if dialerList is null.

```
public void newContact (Contact contact)
```

Adds the new contact `contact` to the dialer list.

Parameter

`contact` the contact being added.

Throws

NullPointerException if `contact` is null.

DialerListNoQueueing

This wrapper dialer list is used by dialers dropping mismatches. It uses a regular dialer list, and resets the patience time of all created contacts to 0. As a result, contacts that cannot be served immediately (mismatches) leave the system without waiting in queue.

```
package umontreal.iro.lecuyer.contactcenters.dialer;  
  
public class DialerListNoQueueing implements DialerList
```

Constructor

```
public DialerListNoQueueing (DialerList list)  
    Constructs a new dialer list with no queueing by using the inner list list.
```

Parameter

`list` the inner dialer list.

Method

```
public DialerList getDialerList()  
    Returns a reference to the internal dialer list.
```

Returns a reference to the internal dialer list.

DialerState

Represents the state of a dialer.

```
package umontreal.iro.lecuyer.contactcenters.dialer;  
  
public class DialerState
```

Constructor

```
protected DialerState (Dialer dialer)
```

Constructs a new dialer state containing the state of the dialer `dialer`.

Parameter

`dialer` the dialer for which the state must be saved.

Method

```
public DialerActionState[] getDialerActionEvents()
```

Returns an array containing a state object for each dialer action event saved.

Returns the array of dialer action events.

DialerActionState

Represents the information needed to scheduled a dialer action event.

```
package umontreal.iro.lecuyer.contactcenters.dialer;  
  
public class DialerActionState
```

Constructors

```
public DialerActionState (Contact contact, boolean success, double  
                           dialEndTime)
```

Constructs a new dialer action event state object for a contact **contact**. The **success** flag gives the success indicator when the dial-up is finished at simulation time **dialEndTime**.

Parameters

contact the contact being dialed.

success the success indicator.

dialEndTime the simulation time of success or failure.

```
public DialerActionState (DialerActionEvent ev)
```

Constructs a new dialer action state object from the dialer action event **ev**.

Parameter

ev the dialer action event to extract information from.

Methods

```
public Contact getContact()
```

Returns the contact being dialed.

Returns the contact being dialed.

```
public boolean isSuccessful()
```

Returns the success indicator of the dial.

Returns the success indicator of the dial.

```
public double getDialEndTime()
```

Returns the time at which success or failure will occur.

Returns the success or failure time.

MismatchChecker

This agent-group listener checks that the number of free agents in the test and target sets for a given dialer never fall outside the user-defined thresholds while dialing is in-progress. This listener is constructed using a dialer using an instance of `ThresholdDialerPolicy` as a dialer's policy. It should then be registered with all agent groups in the target sets.

Each time a service begins (and the number of free agents is reduced), the method `checkThresh()` is called, and checks for the thresholds. If the number of free agents becomes smaller than the given threshold, in-progress dialing is stopped. If the policy is not an instance of `ThresholdDialerPolicy`, this listener does nothing.

```
package umontreal.iro.lecuyer.contactcenters.dialer;

public class MismatchChecker implements AgentGroupListener
```

Constructor

```
public MismatchChecker (Dialer dialer)
```

Constructs a new mismatch checker for the dialer `dialer`.

Parameter

`dialer` the dialer for which mismatches are checked.

Method

```
public void checkThresh()
```

Checks the thresholds on the number of free agents in the test and target sets for the dialer's policy of the associated dialer.

Package `umontreal.iro.lecuyer.contactcenters.router`

Contains the contact routing facilities. A *router*, called an *automatic call distributor* (ACD) for call centers, can be any class listening to new contacts, and assigning them to agent groups or adding them to waiting queues. The router listens to service terminations to assign queued contacts to free agents and to waiting queue events for statistical collection and overflow support.

This package provides the `Router` base class as a basis to implement routers using almost arbitrary policy. It can listen to new contacts and interact with waiting queues and agent groups, which makes it a central point in any contact center. For contacts to be counted correctly during statistical collection, an *exited-contact listener* can also be registered with a router which knows exactly when contacts abandon, are blocked, and are served. Figure 2 gives a UML diagram summarizing how the router is connected to the other parts of the system.

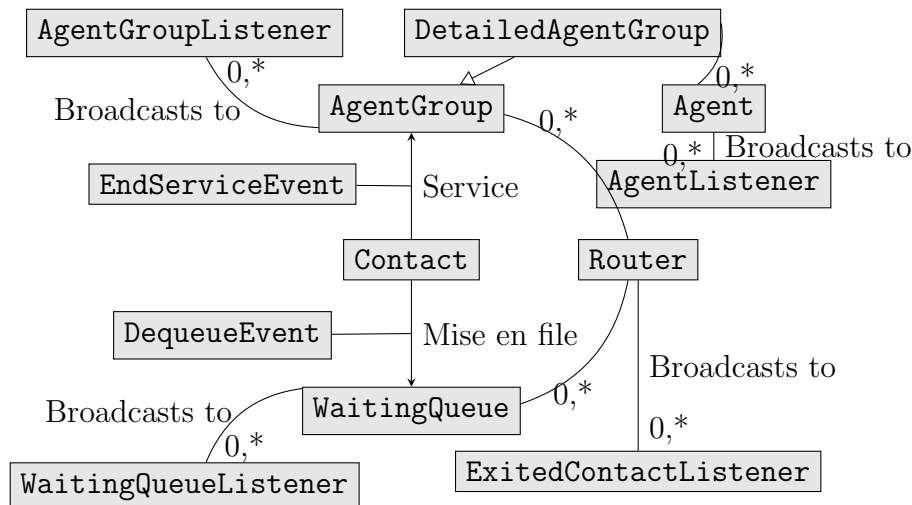


Figure 2: UML diagram describing the routing of contacts

The routing policy itself must be implemented in a subclass by defining fields for the data and implementing or overriding methods for the routing logic. The router needs schemes for agent and contact selections, and it can optionally clear waiting queues when the contact center does not have idle or busy agents capable of serving the waiting contacts. Algorithms to process dequeued and served contacts may also be needed in complex systems supporting overflow or service by multiple agents.

This package provides a few predefined policies inspired from [17] and [12]. These policies do not cover all possible scenarios, but new policies can easily be added.

A first class of policies uses ordered lists as follows. For each contact type k , the *type-to-group map* defines an ordered list $i_{k,0}, i_{k,1}, \dots$ of agent groups. For each agent group i , the *group-to-type map* defines an ordered list $k_{i,0}, k_{i,1}, \dots$ of contact types. These lists indicate which agent groups can serve a contact of type k and which contact types can be served by

agents in group i , respectively. The order of the elements can be used to define priorities. This data structure prevents contact types or agent groups from sharing the same priority, and may produce inconsistent routing policies. For example, a bad router could assign new contacts of type k to agents in group i without pulling contacts of type k from queues when an agent in group i becomes free. Checker methods are provided in `RoutingTableUtils` to detect this problem, but they need to linearly scan the routing tables. As a result, they must be manually called by the user to avoid decreasing the performance.

In a second type of policy, *matrices of ranks* assign ranks or priorities $r_{TG}(k, i)$ and $r_{GT}(i, k)$ to contacts of type k served by agents in group i . If the rank is ∞ , i.e., `Double.POSITIVE_INFINITY`, contacts of type k cannot be served by agents in group i . Otherwise, the smaller is the rank, the higher is the priority of contacts of type k for agents in group i . The matrix defining $r_{TG}(k, i)$ specifies how contacts prefer agents, and is used for agent selection. The second matrix, defining $r_{GT}(i, k)$, specifies how agents prefer contacts, and is used for contact selection. In many cases, it is possible to have $r_{GT}(i, k) = r_{TG}(k, i)$ and specify a single matrix of ranks. This structure allows equal priorities to exist, but routing policies are more complex. When ranks are equal, a secondary algorithm must be used for tie breaking, reducing the performance of the simulator.

The package also supports the *incidence matrix*, which assigns a boolean value $m(i, k)$ for each contact types and agent groups. $m(i, k)$ is *true* if and only if contacts of type k can be served by agents in group i . Such a matrix is not used for routing because it does not encode any priority, but the package provides methods to convert it to a type-to-group, group-to-type, or matrix of ranks.

The package also provides some helper classes and methods to ease the implementation of routers with complex routing policies. These methods can test the consistency of routing information data structures, and perform conversions from one structure to another. They can also help in contact and agent selections.

Router

Represents a contact router which can perform agent and contact selections. A router links the contact sources, agent groups and waiting queues together and acts as a central element of the contact center. It supports a certain number of contact types and contains slots for waiting queues and agent groups.

Dequeued contacts and freed agents are notified to the router through nested classes implementing the appropriate listener interfaces. These classes listen to the connected agent groups and waiting queues only. Agent groups and waiting queues are connected to the router using the `setAgentGroup (int, AgentGroup)` and `setWaitingQueue (int, WaitingQueue)` methods, respectively. During connection, they are assigned a numerical identifier to be referred to efficiently during routing.

This abstract class does not implement any routing policy. To implement such a policy, many informations must be provided in a subclass: data structures, algorithms for agent, waiting queue and contact selections, and an algorithm to automatically clear waiting queues. A router can also specify what happens when a contact is served or abandons. We now examine these elements in more details. Data structures are encoded into fields, and usually consist of a type-to-group and a group-to-type maps, or matrices of ranks. Algorithms are provided by overriding methods.

When a new contact is notified through its `newContact (Contact)` method specified by the `NewContactListener` interface, the router performs *agent selection*, i.e., it tries to assign an agent to the contact. The `selectAgent (Contact)` method is used to select the agent, and `selectWaitingQueue (Contact)` is called if no free agent is available.

The router supports contact rerouting which works as follows. When a contact is queued, the router gets a delay using `getReroutingDelay (DequeueEvent, int)`. If that delay is finite and greater than or equal to 0, a *rerouting event* is scheduled. When such an event happens, the router tries to use `selectAgent (DequeueEvent, int)` to assign an agent to a queued contact. If this rerouting fails, the router uses `selectWaitingQueue (DequeueEvent, int)` to decide if the contact should be dropped, transferred into another queue, or kept in the same queue. After this queue reselection has happened, the router uses `getReroutingDelay (DequeueEvent, int)` again to decide if a subsequent rerouting will happen. By default, this functionality is disabled. One has to override `getReroutingDelay (DequeueEvent, int)` and `selectAgent (DequeueEvent, int)` to use rerouting.

When an agent becomes free, the router must perform *contact selection*, i.e., it must try to assign a queued contact to the free agent through the `checkFreeAgents (AgentGroup, Agent)` method. The `checkFreeAgents (AgentGroup, Agent)` method is called, and usually calls `selectContact (AgentGroup, Agent)` to get queued contacts. If no queued contact is available for the free agent, the agent remains free.

The router also supports agent rerouting which works as follows. If an agent has finished the service of a contact and cannot find a new contact to serve, before letting the agent idle, the router gets a delay using `getReroutingDelay (Agent, int)`. If this delay is finite and greater than or equal to 0, the router schedules an event that will try to assign a

new contact to the agent. The contact is selected using the `selectContact (Agent, int)` method. As with contact rerouting, agent rerouting can happen multiple times and it is disabled by default. One needs to use detailed agent groups considering individual agents and override `getReroutingDelay (Agent, int)` as well as `selectContact (Agent, int)` to take advantage of agent rerouting.

At some moments during the day, queued contacts may never be served, because no skilled agent is present. For example, when the center closes, all agents leave and queued contacts are forced to wait forever or abandon. To avoid this, an additional algorithm may be implemented in `checkWaitingQueues (AgentGroup)` to automatically clear the queues when no agent can serve contacts. This clearing is disabled by default but can be enabled by using `setClearWaitingQueue (int, boolean)` or `setClearWaitingQueues (boolean)`.

Finally, the moment a contact exits can be controlled. By default, dequeued and served contacts exit the system, but it is possible to override methods in this class to change this behavior, e.g., transfer a dequeued contact to another queue, transfer a served contact to another agent, etc.

Note that the blocking, dequeue, end-contact and end-service indicators `Integer.MAX_VALUE - 1000` through `Integer.MAX_VALUE` are reserved for present and future use by routers. Dequeue type 0 is also reserved, and represents the beginning of the service for a queued contact. The constant `DEQUEUE_TYPE_BEGIN_SERVICE` can be used to represent this.

Note: the `ExitedContactListener` implementations are notified in the order of the list returned by `getExitedContactListeners()`, and an exited-contact listener modifying the list of listeners by using `addExitedContactListener (ExitedContactListener)` or `removeExitedContactListener (ExitedContactListener)` could result in unpredictable behavior.

```
package umontreal.iro.lecuyer.contactcenters.router;
```

```
public abstract class Router implements NewContactListener
```

Fields

```
public static final int BLOCKTYPE_NOLINE
```

Contact blocking type occurring when there is no communication channel available in the trunk group associated with an incoming contact.

```
public static final int BLOCKTYPE_QUEUEFULL
```

Contact blocking type occurring when the total queue capacity is exceeded upon the arrival of a contact.

```
public static final int BLOCKTYPE_CANTQUEUE
```

Contact blocking type occurring when a contact cannot be queued, i.e., `selectWaitingQueue (Contact)` returns null.

```
public static final int DEQUEUEUETYPE_BEGINSERVICE
```

Contact dequeueing type representing the beginning of the service.

```
public static final int DEQUEUEUETYPE_NOAGENT
```

Contact dequeueing type occurring when a waiting queue is cleared because there is no agent in the system capable of serving the contact.

```
public static int DEQUEUEUETYPE_FANTOM
```

Contact dequeue type used to remove multiple copies of a contact from waiting queues. When a contact has to wait in more than one waiting queues, it can exit any of these queues at any time. When the contact is dequeued, e.g., because it is transferred to an agent. In this case, the contact also needs to be removed from other queues. This dequeue type can be used to avoid such contacts being counted several times by statistical facilities.

```
public static int DEQUEUEUETYPE_TRANSFER
```

Contact dequeue type used when transferring a contact from a waiting to another waiting queue.

```
protected int dqTypeRet
```

Contains the dequeue type used when a contact leaves a queue to enter a new one. By default, this is set to 1.

Constructor

```
public Router (int numTypes, int numQueues, int numGroups)
```

Constructs a new router with `numTypes` contact types, `numQueues` waiting queues, and `numGroups` agent groups.

Parameters

`numTypes` number of contact types.

`numQueues` number of waiting queues.

`numGroups` number of agent groups.

Throws

`IllegalArgumentException` if any argument is negative.

Methods

```
public boolean isKeepingReroutingEvents()
```

Determines if this router keeps track of all rerouting events scheduled. By default, these events are discarded, i.e., they are stored in the event list only.

Returns `true` if the router keeps track of the rerouting events, `false` otherwise.

```
public void setKeepingReroutingEvents (boolean keep)
```

Sets the keep-rerouting-events indicator to `keep`.

Parameter

`keep` the value of the indicator.

```
public Iterator<ContactReroutingEvent> contactReroutingEventsIterator  
( )
```

Constructs and returns an iterator for the contact rerouting events. If `isKeepingReroutingEvents()` returns `true`, the iterator is constructed from the set returned by `getContactReroutingEvents()`. Otherwise, an iterator traversing the event list and filtering the appropriate events is constructed and returned.

Returns the iterator for contact rerouting events.

```
public Map<DequeueEvent, ContactReroutingEvent> getContactReroutingEvents  
( )
```

Returns an unmodifiable map containing the currently scheduled contact rerouting events. Each key of this map corresponds to a dequeue event while each value corresponds to an instance of `ContactReroutingEvent`. If rerouting events are not kept, this throws an `IllegalStateException`.

Returns the map of contact rerouting events.

Throws

`IllegalStateException` if rerouting events are not kept.

```
public Iterator<AgentReroutingEvent> agentReroutingEventsIterator  
( )
```

Constructs and returns an iterator for the agent rerouting events. If `isKeepingReroutingEvents()` returns `true`, the iterator is constructed from the set returned by `getAgentReroutingEvents()`. Otherwise, an iterator traversing the event list and filtering the appropriate events is constructed and returned.

Returns the iterator for agent rerouting events.

```
public Map<Agent, AgentReroutingEvent> getAgentReroutingEvents  
( )
```

Returns an unmodifiable map containing the currently scheduled agent rerouting events. Each key of this map corresponds to an `Agent` object while each value corresponds to an instance of `AgentReroutingEvent`. If rerouting events are not kept, this throws an `IllegalStateException`.

Returns the map of agent rerouting events.

Throws

`IllegalStateException` if rerouting events are not kept.

```
public RouterState save()
```

Saves the state of this router, and returns the resulting state object.

Returns the current state of this router.

```
public void restore (RouterState state)
```

Restores the state `state` of this router.

Parameter

`state` the saved state of the router.

```
public int getTotalQueueCapacity()
```

Returns the total capacity of the waiting queues for this router. This capacity determines the maximal number of contacts that can be queued simultaneously by this router. By default, this is `Integer.MAX_VALUE`, i.e., infinite.

Returns the total queue capacity of the router.

```
public void setTotalQueueCapacity (int capacity)
```

Sets the total queue capacity to `capacity` for this router. If the given capacity is negative, an `IllegalArgumentException` is thrown. If the capacity is less than the total number of queued contacts, this throws an `IllegalStateException`.

Parameter

`capacity` the new total queue capacity.

Throws

`IllegalArgumentException` if the given capacity is negative.

`IllegalStateException` if the given capacity is less than the actual number of queued contacts.

```
public int getCurrentQueueSize()
```

Returns the total number of contacts in the connected waiting queues.

Returns the total number of contacts in queues.

```
public int getNumContactTypes()
```

Returns the number of contact types supported by this router.

Returns the supported number of contact types.

```
public int getNumAgentGroups()
```

Returns the number of agent groups supported by this router.

Returns the number of agent groups.

```
public int getNumWaitingQueues()
```

Returns the number of waiting queues supported by this router.

Returns the number of waiting queues.

```
public WaitingQueue getWaitingQueue (int q)
```

Returns the waiting queue with index `q` for this router. If `q` is less than 0 or greater than or equal to the number of supported queues, an exception is thrown. Calling the `WaitingQueue.getId()` method on the returned waiting queue should return `q`, unless this method returns `null`.

Parameter

`q` the index of the queue.

Returns the associated waiting queue, or `null` if no queue is defined for this index.

Throws

`IndexOutOfBoundsException` if `q` is negative or greater than or equal to `getNumWaitingQueues()`.

```
public WaitingQueue[] getWaitingQueues()
```

Returns an array containing the waiting queues attached to this router.

Returns the waiting queues attached to this router.

```
public void setWaitingQueue (int q, WaitingQueue queue)
```

Associates the waiting queue `queue` with the index `q` in the router. The method tries to set the queue id to `q` and registers a waiting-queue listener for `queue` to be notified about automatic dequeues if needed. If a waiting queue was previously associated with the index, the router's waiting-queue listener is removed from that previous waiting queue.

Note that some routers assume that waiting queues use FIFO discipline. In this case, one should use `StandardWaitingQueue` instances only. Using `PriorityWaitingQueue` may lead to routing not corresponding to the defined policy.

Parameters

`q` the index of the queue.

`queue` the queue to be associated.

Throws

`IllegalStateException` if the queue id was already set to another value than `q`.

`IndexOutOfBoundsException` if `q` is negative or greater than or equal to `getNumWaitingQueues()`.

```
public boolean mustClearWaitingQueue (int q)
```

Determines if the router must clear the waiting queue `q` when all queued contacts cannot be served since no agent capable of serving them is online anymore. By default, this is set to `false`.

Parameter

`q` the index of the checked waiting queue.

Returns the clear-waiting-queue indicator.

Throws

`IndexOutOfBoundsException` if `q` is negative or greater than or equal to `getNumWaitingQueues()`.

```
public void setClearWaitingQueue (int q, boolean b)
```

Sets the clear-waiting-queue indicator for the waiting queue `q` to `b`. See `mustClearWaitingQueue (int)` for more information.

Parameters

`q` the index of the affected waiting queue.

`b` the new value of the indicator.

Throws

`IndexOutOfBoundsException` if `q` is negative or greater than or equal to `getNumWaitingQueues()`.

See also `mustClearWaitingQueue (int)`

```
public void setClearWaitingQueues (boolean b)
```

Sets the clear-waiting-queue indicator to `b` for all waiting queues. See `mustClearWaitingQueue (int)` for more information.

Parameter

`b` the new value of the indicator.

See also `mustClearWaitingQueue (int)`

```
public SingleTypeContactFactory getContactFactory (int k)
```

Returns the contact factory used by the simulator to create contacts of type `k`. This factory may be used by some routing policies to obtain information such as the distribution of service times. When a routing policy uses this information, the simulator should create contacts of type `k` with this single-type contact factory only.

Parameter

`k` the contact type identifier.

Returns the contact factory.

```
public void setContactFactory (int k, SingleTypeContactFactory factory)
```

Sets the contact factory used to create contacts of type `k` to `factory`.

Parameters

k the contact type identifier.

factory the contact factory.

```
public AgentGroup getAgentGroup (int i)
```

Returns the agent group with index *i* for this router. If *i* is less than 0 or greater than or equal to the number of groups, an exception is thrown. Calling `AgentGroup.getId()` on the returned group should return *i*, unless this method returns `null`.

Parameter

i the index of the agent group.

Returns the associated agent group, or `null` if no agent group is defined for this index.

Throws

`IndexOutOfBoundsException` if *i* is negative or greater than or equal to `getNumAgentGroups()`.

```
public AgentGroup[] getAgentGroups()
```

Returns an array containing the agent groups attached to this router.

Returns the attached agent groups.

```
public void setAgentGroup (int i, AgentGroup group)
```

Associates the agent group *group* with the index *i* in the router. The method tries to set the identifier of the group to *i* and registers an agent-group listener to be notified about agents becoming free in order to perform contact selection. If an agent group was previously associated with the index, the router's agent-group listener is removed from that previous agent group.

Parameters

i the index of the agent group.

group the agent group to be associated.

Throws

`IllegalStateException` if the group id was already set to another value than *i*.

`IndexOutOfBoundsException` if *i* is negative or greater than or equal to `getNumAgentGroups()`.

```
public List<Dialer> getDialers (int i)
```

Returns a list containing the dialers which will be triggered when the service of a contact by an agent in group *i* ends. This list, which may contain only non-`null` instances of the `Dialer` class, should be used instead of an agent-group listener to activate the dialer. As opposed to an agent-group listener requesting dialers to try calls, dialers in the returned list are activated only after contact selection for agents in group *i* is done, and they are guaranteed to be activated in the order given by the list.

Parameter

i the index of the agent group.

Returns the list of dialers.

Throws

`ArrayIndexOutOfBoundsException` if the agent group index is out of bounds.

```
public abstract boolean canServe (int i, int k)
```

Returns `true` if and only if some agents in group *i* are authorized to serve contacts of type *k* by this router.

Parameters

i the agent group index.

k the contact type index.

Returns determines if contacts can be served.

```
public boolean needsDetailedAgentGroup (int i)
```

Determines if the agent group *i* should consider individual agents. This does not determine directly how the agent group returned by `getAgentGroup (int)` is implemented. This method only gives clues to a simulator on how to construct the concerned agent group.

Parameter

i the index of the agent group.

Returns the detailed status of the agent group.

```
public WaitingQueueType getWaitingQueueType()
```

Returns an indicator describing how the implemented routing policies organizes waiting queues. The supported modes of organization cover the most common cases only: waiting queues corresponding to contact types or agent groups. For any other modes, the `WaitingQueueType.GENERAL` must be used.

By default, this method returns `WaitingQueueType.GENERAL`.

Returns the organization mode of waiting queues.

```
public WaitingQueueStructure getNeededWaitingQueueStructure  
(int q)
```

Returns the needed data structure for waiting queue with index *q*. This method is used by the simulator to get clues on how to construct the waiting queue; it does not affect directly the implementation of the waiting queue returned by `getWaitingQueue (int)`. By default, this returns `WaitingQueueStructure.LIST`.

Parameter

q the index of the waiting queue.

Returns the structure indicator.

```
public Comparator<? super DequeueEvent> getNeededWaitingQueueComparator
(int q)
```

Determines how contacts in queue should be compared with each other for waiting queue `q`. This comparator is used by a simulator to construct a waiting queue if `getNeededWaitingQueueStructure (int)` returns `WaitingQueueStructure.SORTEDSET` or `WaitingQueueStructure.PRIORITY`. By default, this returns `null`.

Parameter

`q` the index of the waiting queue.

Returns the waiting queue comparator.

```
public void newContact (Contact contact)
```

This method is called when the new contact `contact` enters in the system and should not be overridden. The `Contact.setRouter (Router)` method is first used to set the router of the new contact to this object. Then, if `Contact.getTrunkGroup()` returns a non-`null` value, a communication channel is allocated. If no communication channel is available for the contact, the contact is blocked with blocking type `BLOCKTYPE_NOLINE`. If the contact has no associated trunk group or if a communication channel could be successfully allocated, the `selectAgent (Contact)` method is called to try to assign it an agent. In case of failure, i.e., `selectAgent (Contact)` returns `null`, the router tries to queue the contact. If the total queue size is equal to the total queue capacity, or if `selectWaitingQueue (Contact)` returns `null`, the contact is blocked with blocking type `BLOCKTYPE_QUEUEFULL` or `BLOCKTYPE_CANTQUEUE`, respectively.

Parameter

`contact` the arrived contact.

Throws

`IllegalStateException` if `Contact.getRouter()` returns a non-`null` value before the router is set.

`IllegalArgumentException` if the contact type identifier of the contact is negative or greater than or equal to `getNumContactTypes()`.

```
public void addExitedContactListener (ExitedContactListener listener)
```

Adds the exited-contact listener `listener` to this router. If the listener is already registered, nothing happens.

Parameter

`listener` the listener being added.

Throws

`NullPointerException` if `listener` is `null`.

```
public void removeExitedContactListener (ExitedContactListener listener)
```

Removes the exited-contact listener `listener` from this router.

Parameter

listener the exited contact listener being removed.

```
public void clearExitedContactListeners()
```

Removes all the exited-contact listeners registered to this router.

```
public List<ExitedContactListener> getExitedContactListeners()  
( )
```

Returns an unmodifiable list containing all the exited-contact listeners registered with this router.

Returns the list of all registered exited-contact listeners.

```
public void notifyBlocked (Contact contact, int bType)
```

Notifies every registered listener that the contact **contact** was blocked with blocking type **bType**.

Parameters

contact the blocked contact.

bType the blocking type.

```
public void notifyDequeued (DequeueEvent ev)
```

Notifies every registered listener that a contact left the waiting queue, this event being represented by **ev**.

Parameter

ev the event representing the contact having left the queue.

```
public void notifyServed (EndServiceEvent ev)
```

Notifies every registered listener that a contact was served, the service being represented by the end-service event **ev**.

Parameter

ev the end-service event representing the end of the service.

```
public void exitServed (EndServiceEvent ev)
```

This method must be called to notify a contact exiting the system after an end of service with end-service event **ev**. It notifies any registered exited-contact listener, and releases the communication channel taken by the contact. This must be called after the communication between the contact and an agent, before after-contact work.

Parameter

ev the end-service event.

```
public void exitDequeued (DequeueEvent ev)
```

This method must be called to notify that a contact exited the system after being dequeued, **ev** representing the dequeue event. It notifies any registered exited-contact listener, and releases the communication channel taken by the contact.

Parameter

ev the dequeue event.

```
public void exitBlocked (Contact contact, int bType)
```

This method can be called when the contact **contact** was blocked by the router with blocking type **bType**. It notifies any registered exited-contact listener, and releases the communication channel taken by the contact. The **bType = BLOCKTYPE.NOLINE** value is reserved for the special case where there is no available communication channel for the contact.

Parameters

contact the contact being blocked.

bType the blocking type.

```
protected void startDialers (AgentGroup group)
```

Starts the dialers after the service of a contact by an agent in group **group**. This method is called after **checkFreeAgents (AgentGroup, Agent)** and should call the **Dialer.dial()** method on one or more dialers. The default implementation starts all the dialers in the list **getDialers (group.getId())**.

Parameter

group the agent group being notified.

```
public void init()
```

This method is called at the beginning of the simulation to reset the state of this router.

```
protected abstract EndServiceEvent selectAgent (Contact contact)
```

Begins the service of the contact **contact** by trying to assign it a free agent. The method must select an agent group with a free agent (or a specific free agent), start the service, and return the end-service event if the service was started, or **null** otherwise.

Parameter

contact the contact being routed to an agent.

Returns the end-service event representing the started service, or `null` if the contact could not be served immediately.

```
protected EndServiceEvent selectAgent (DequeueEvent dqEv, int
                                         numReroutingsDone)
```

Selects an agent for serving a queued contact in the context of rerouting. The event `dqEv` is used to represent the dequeued contact, while `numReroutingsDone` indicates the number of reroutings that has happened so far. The method should return the end-service event corresponding to the contact's new service by an agent, or `null` for the contact to stay in queue.

Parameters

`dqEv` the dequeue event representing the queued contact.

`numReroutingsDone` the number of preceding reroutings.

Returns the end-service event, or `null`.

```
protected abstract DequeueEvent selectWaitingQueue (Contact contact)
```

Selects a waiting queue and puts the contact `contact` into it. Returns the dequeue event if the contact could be queued, or `null` otherwise.

Parameter

`contact` the contact being queued.

Returns the dequeue event representing the queued contact, or `null` if the contact could not be queued.

```
protected DequeueEvent selectWaitingQueue (DequeueEvent dqEv, int
                                         numReroutingsDone)
```

Selects a waiting queue for a queued contact in the context of rerouting. The event `dqEv` is used to represent the queued contact, while `numReroutingsDone` indicates the number of reroutings that has happened so far. The method should return the dequeue event corresponding to the contact's new queue, or `null` if the contact is required to leave the system. If no transfer of queue is required, this method should return `dqEv`. If a transfer occurs, one can use the `dqTypeRet` field to store the dequeue type of the contact leaving the queue.

Parameters

`dqEv` the dequeue event representing the queued contact.

`numReroutingsDone` the number of preceding reroutings.

Returns the dequeue event, or `null`.

`protected boolean checkFreeAgents (AgentGroup group, Agent agent)`

This method is called when the agent `agent` in agent group `group` becomes free. If the given agent is `null`, the method assumes that one or more arbitrary agents in the group became free. The method must select a contact to be transferred to the free agent. The selected contacts come from waiting queues, and must be removed from the queues with dequeue type `DEQUEUEUETYPE_BEGINSERVICE` before they are transferred to agents. The method returns `true` if and only if at least one free agent could be made busy.

The default implementation calls `selectContact (AgentGroup, Agent)` to get a new dequeue event representing the removed contact, extracts the contact, and routes it to an agent, until `group` has no more free agent.

Parameters

`group` the affected agent group.

`agent` the agent having ended its service.

Returns `true` if some free agents became busy, `false` otherwise.

`protected DequeueEvent selectContact (AgentGroup group, Agent agent)`

Returns a dequeue event representing a queued contact to be served by the agent `agent` in agent group `group`. If `agent` is `null`, the method must return a contact that can be served by any agent in the group. If no contact is available, this method returns `null`. The selected contacts come from waiting queues attached to the router. Before the selected contact is returned, it must be removed from its queue with dequeue type `DEQUEUEUETYPE_BEGINSERVICE`, e.g., by using `queue.removeFirst (DEQUEUEUETYPE_BEGINSERVICE)`, or `queue.remove (ev, DEQUEUEUETYPE_BEGINSERVICE)`, etc.

Generally, it is sufficient to override this method instead of `checkFreeAgents (AgentGroup, Agent)`. One can override `checkFreeAgents (AgentGroup, Agent)` to improve efficiency when looking for contacts in the same waiting queue. This method is not abstract and returns `null` by default in order to allow `checkFreeAgents (AgentGroup, Agent)` to be overridden without implementing this method.

Parameters

`group` the affected agent group.

`agent` the agent having ended its service.

Returns the dequeue event representing the contact being selected.

`protected DequeueEvent selectContact (Agent agent, int numReroutingsDone)`

Selects a new contact for the agent `agent`, in the context of rerouting.

Parameters

`agent` the affected agent.

`numReroutingsDone` the number of preceding reroutings.

Returns the selected contact, or `null`.

`protected abstract void checkWaitingQueues (AgentGroup group)`

This method is called when the agent group `group` contains no more online agents, i.e., `AgentGroup.getNumAgents()` returns 0. It must check each waiting queue accessible for agents in this group to determine if they need to be cleared. A queue is cleared if no agent, whether free or busy, is available to serve any contact in it.

Parameter

`group` the agent group with no more agents.

`protected double getReroutingDelay (DequeueEvent dqEv, int numReroutingsDone)`

Returns the delay, in simulation time units, after which a queued contact should be rerouted. The value of `numReroutingsDone` gives the number of preceding reroutings, and `dqEv` is the dequeue event. If this delay is negative, infinite, or NaN, no rerouting happens for the contact. `numReroutings` will be -1 when this method is called at the time the contact is queued. By default, this method returns `Double.POSITIVE_INFINITY`.

Parameters

`dqEv` the dequeue event representing the queued contact.

`numReroutingsDone` the number of reroutings so far.

Returns the rerouting delay.

`protected double getReroutingDelay (Agent agent, int numReroutingsDone)`

Returns the delay, in simulation time units, after which an agent `agent` should try a new time to get a contact to serve. If no rerouting should happen, the returned delay must be negative or NaN. `numReroutings` will be -1 when this method is called at the end of a service. By default, this method returns `Double.POSITIVE_INFINITY`.

Parameters

`agent` the idle agent, or `null`.

`numReroutingsDone` the number of previous reroutings for the agent.

Returns the rerouting delay.

`protected void beginService (EndServiceEvent ev)`

This method is called when the service of a contact, represented by the event `ev`, begins. By default, this method does nothing.

Parameter

`ev` the end-service event.

`protected void endContact (EndServiceEvent ev)`

This method is called when the communication between a contact and an agent is finished. By default, it calls `exitServed (EndServiceEvent)`.

Parameter

`ev` the end-service event.

`protected void endService (EndServiceEvent ev)`

This method is called when the service (communication and after-contact work) of a contact in an agent group has ended. By default, this does nothing.

Parameter

`ev` the end-service event.

`protected void enqueued (DequeueEvent ev)`

This method is called when a contact is enqueued, `ev` representing the dequeue event. By default, this method does nothing.

Parameter

`ev` the dequeue event.

`protected void dequeued (DequeueEvent ev)`

This method is called when a contact leaves a waiting queue, `ev` representing the corresponding dequeue event. By default, for any effective dequeue type other than 0, this calls `exitDequeued (DequeueEvent)`. This method should not notify an exiting contact for a 0 dequeue type since it is reserved for queued and served contacts.

Parameter

`ev` the dequeue event.

`public String formatWaitingQueues()`

Formats the connected waiting queues as a string. For each queue slot, the returned string contains a line with the text `Waiting queue q:` followed by the queue's `toString` result. If no queue is connected to the slot, `undefined` is used as the waiting queue descriptor.

Returns the waiting queues of the router.

`public String formatAgentGroups()`

Formats the connected agent groups as a string. For each group slot, the returned string contains a line with the text `Agent group i:` followed by the group's `toString` result. If no group is connected to the slot, `undefined` is used as the agent group descriptor.

Returns the agent groups of the router.

`public WaitingQueueListener getWaitingQueueListener()`

Returns the waiting-queue listener registered with each waiting queue connected to this router. Obtaining this listener can be useful to replace it, in the list of listeners of a waiting queue, by a wrapper executing code before or after some events.

Returns the waiting-queue listener registered with each waiting queue.

```
public AgentGroupListener getAgentGroupListener()
```

Returns the agent-group listener registered with each agent group connected to this router. Obtaining this listener can be useful to replace it, in the list of listeners of an agent group, by a wrapper executing code before or after some events.

Returns the agent-group listener registered with each agent group.

```
public String toLongString()
```

Returns a string representation of detailed information about the router. This returns a string representation of each associated waiting queue and agent group and routing policies. For a short, one-line description, `toString()` should be used.

Returns a string representation of detailed information about the router.

WaitingQueueType

Represent possible roles of waiting queues for routing policies.

```
package umontreal.iro.lecuyer.contactcenters.router;  
  
public enum WaitingQueueType
```

Constants

CONTACTTYPE

When this type is used, there must be one waiting queue for each contact type. More specifically, waiting queue k , for $k = 0, \dots, K - 1$, contains only contacts of type k , where K is the total number of contact types.

AGENTGROUP

When this type is used, there must be one waiting queue for each agent group. More specifically, waiting queue i , for $i = 0, \dots, I - 1$, contains only contacts that are to be served by agents in group i , where I is the total number of agent groups.

GENERAL

Used when the waiting queues do not correspond to the schemes described by **CONTACTTYPE** or **AGENTGROUP**.

WaitingQueueStructure

Possible data structures for waiting queues.

```
package umontreal.iro.lecuyer.contactcenters.router;  
  
public enum WaitingQueueStructure
```

Constants

LIST

Queued contacts are placed in an ordinary list, in the order they enter the queue. When an agent becomes free, the first contact is usually removed from the queue. This structure therefore implements a FIFO queue. This is the most common, the fastest and is the default structure.

PRIORITY

Queued contacts are put into a priority queue, usually implemented using a heap. A comparator is used to sort contacts by priority. A free agent removes the contacts with the highest priority first. However, if a priority queue is scanned, the order of the contacts might not be the order imposed by the comparator. The structure used only guarantees that the first contact is the “smallest” with respect to the comparator given.

SORTEDSET

Queued contacts are put into a sorted set, usually implemented by a binary tree. This is similar to **PRIORITY**, but the contacts in queue can be enumerated in the correct order at any time. However, sorted sets are slower than priority queues.

ContactReroutingEvent

Represents an event happening when the router tries to reroute a queued contact to an agent, or another queue.

```
package umontreal.iro.lecuyer.contactcenters.router;  
  
public final class ContactReroutingEvent extends Event
```

Constructor

```
public ContactReroutingEvent (Router router, DequeueEvent dqEv, int  
                             numReroutingsDone)
```

Constructs an event that will reroute the queued contact `dqEv` to an agent or another queue.

Parameters

`router` the router this event is linked to.

`dqEv` the dequeue event.

`numReroutingsDone` the number of reroutings done.

Methods

```
public Router getRouter()
```

Returns the router associated with this event.

Returns the associated router.

```
public DequeueEvent getDequeueEvent()
```

Returns the dequeue event associated with this rerouting event.

Returns the associated dequeue event.

```
public int getNumReroutingsDone()
```

Returns the number of reroutings done, i.e., the number of calls to `actions()` having resulted in the contact not being transferred to an agent.

Returns the number of reroutings that has happened.

AgentReroutingEvent

Represents an event happening when the router tries once more to affect a contact to an agent.

```
package umontreal.iro.lecuyer.contactcenters.router;  
  
public final class AgentReroutingEvent extends Event
```

Constructor

```
public AgentReroutingEvent (Router router, Agent agent, int  
                           numReroutingsDone)
```

Constructs a new agent rerouting event instructing the router **router** to try to find a queued contact for the idle agent **agent** after there was **numReroutingsDone** preceding reroutings.

Parameters

router the router to be used.

agent the agent to be rerouted.

numReroutingsDone the number of preceding trials.

Methods

```
public Router getRouter()
```

Returns the router associated with this event.

Returns the associated router.

```
public Agent getAgent()
```

Returns the agent to be assigned a queued contact.

Returns the agent to be assigned a queued contact.

```
public int getNumReroutingsDone()
```

Returns the number of preceding reroutings.

Returns the number of reroutings already tried.

QueuePriorityRouter

This skill-based router with queue priority ranking is based on the routing heuristic in [12], extended to support queueing. When a contact arrives to the router, an ordered list (the type-to-group map) is used to determine which agent groups are able to serve it, and the order in which they are checked. If agent group $i_{k,0}$ contains at least one free agent, this agent serves the contact. Otherwise, the router tries to test agent groups $i_{k,1}$, $i_{k,2}$, etc. until a free agent is found, or the list of agent groups is exhausted. In other words, the contact *overflows* from one agent group to another. If no agent group in the ordered list associated with the contact's type is able to serve the contact, the contact is inserted into a waiting queue corresponding to its type unless the queue is full. If the total queueing capacity of the router is exceeded, the contact is blocked.

When an agent becomes free, it uses another ordered list (the group-to-type map) to determine which types of contacts it can serve. If the queue containing contacts of type $k_{i,0}$ is non-empty, the first contact, i.e., the contact of type $k_{i,0}$ with the longest waiting time, is removed and handled to the free agent. Otherwise, the queues containing contacts of types $k_{i,1}$, $k_{i,2}$, etc. are queried similarly for contacts to be served. If no contact is available in any accessible waiting queue, the agent stays free. The router behaves as if a priority queue was associated with each agent group, implementing priorities by using several FIFO waiting queues.

This router should be used only when the type-to-group and group-to-type maps are specified as input data. If one table has to be generated from the other one, the induced arbitrary order of the lists can affect the performance of the contact center.

```
package umontreal.iro.lecuyer.contactcenters.router;
```

```
public class QueuePriorityRouter extends Router
```

Fields

```
protected int[][] typeToGroupMap
```

Contains the type-to-group map routing table.

```
protected int[][] groupToTypeMap
```

Contains the group-to-type map routing table.

Constructor

```
public QueuePriorityRouter (int[][] typeToGroupMap, int[][] groupToTypeMap)
```

Constructs a new queue priority router with a type-to-group map `typeToGroupMap`, and a group-to-type map `groupToTypeMap`.

Parameters

`typeToGroupMap` the type-to-group map.

`groupToTypeMap` the group-to-type map.

Methods

```
public int[] [] getTypeToGroupMap()
```

Returns the type-to-group map associated with this router.

Returns the associated type-to-group map.

```
public int[] getTypeToGroupMap (int k)
```

Returns the ordered list concerning contact type `k` in the type-to-group map.

Parameter

`k` the index of the contact type.

Returns the ordered list.

```
public int[] [] getGroupToTypeMap()
```

Returns the group-to-type map associated with this router.

Returns the associated group-to-type map.

```
public int[] getGroupToTypeMap (int i)
```

Returns the ordered list concerning agent group `i` in the group-to-type map.

Parameter

`i` the index of the agent group.

Returns the ordered list.

```
public void setRoutingTable (int[] [] typeToGroupMap, int[] []  
                             groupToTypeMap)
```

Changes the routing table for this router. The routing table must be specified using `typeToGroupMap` and `groupToTypeMap`.

Parameters

`typeToGroupMap` the type-to-group map.

`groupToTypeMap` the group-to-type map.

Throws

IllegalArgumentException if the type-to-group map does not contain K rows, or the group-to-type map does not contain I rows.

protected void checkWaitingQueues (AgentGroup group)

This default implementation is suitable only for routers specifying a type-to-group and a group-to-type map and using one waiting queue for each contact type. If the tables are not specified or the number of supported waiting queues is different from the number of supported contact types, this implementation does nothing.

Parameter

group the agent group with no more agents.

public String formatTypeToGroupMap()

Calls `RoutingTableUtils.formatTypeToGroupMap (int[][])` with the type-to-group map associated with this router.

Returns the type-to-group map, formatted as a string.

public String formatGroupToTypeMap()

Calls `RoutingTableUtils.formatGroupToTypeMap (int[][])` with the group-to-type map associated with this router.

Returns the group-to-type map, formatted as a string.

QueueAtLastGroupRouter

This router uses a queue-at-last-group policy. When a new contact of type k arrives, the serving agent is selected the same way as with queue priority routing policy: each agent group $i_{k,0}, i_{k,1}, \dots$ of the type-to-group map is tested to find a free agent. However, if no agent can serve the contact, the contact is put into a waiting queue associated with the last agent group in the ordered list rather than the contact type. As usual, if the router's queue capacity is exceeded, the contact is blocked. When an agent requests a new contact to be served, it looks into its associated waiting queue only. If no contact is available in that queue, the agent remains free. The loss-delay approximation, presented in [2], assumes that the contact center uses this policy.

```
package umontreal.iro.lecuyer.contactcenters.router;
```

```
public class QueueAtLastGroupRouter extends Router
```

Fields

```
public static final int NONE
```

Agent group which is neither loss nor delay.

```
public static final int LOSS
```

Agent group is a loss station; see `isLoss (int, int)`.

```
public static final int DELAY
```

Agent group is a delay station; see `isDelay (int, int)`.

```
public static final int LOSSDELAY
```

Agent group is a loss and delay station.

```
protected int[] [] typeToGroupMap
```

Contains the type-to-group map routing table.

Constructor

```
public QueueAtLastGroupRouter (int numGroups, int[] [] typeToGroupMap)
```

Constructs a new queue at last group router with a type-to-group map `typeToGroupMap`.

Parameters

`numGroups` the number of agent groups.

`typeToGroupMap` the type-to-group map.

Methods

```
public int[] [] getTypeToGroupMap()
```

Returns the type-to-group map associated with this router.

Returns the associated type-to-group map.

```
public int[] getTypeToGroupMap (int k)
```

Returns the ordered list concerning contact type `k` in the type-to-group map.

Parameter

`k` the index of the contact type.

Returns the ordered list.

```
public void setTypeToGroupMap (int[] [] tg)
```

Sets the type-to-group map associated with this router to `tg`.

Parameter

`tg` the new type-to-group map.

```
public int getAgentGroupType (int k, int i)
```

Determines the type of agent group `i` for contacts of type `k`. This returns `LOSS` if the group is a loss station, i.e., `isLoss (int, int)` returns `true`, `DELAY` if it is a delay station (`isDelay (int, int)` returns `true`), and `NONE` otherwise.

Parameters

`k` the contact type.

`i` the agent group.

Returns the status of agent group for the contact type.

Throws

`IndexOutOfBoundsException` if `i` or `k` are negative, `i` is greater than or equal to `Router.getNumAgentGroups()` or `k` is greater than or equal to `Router.getNumContactTypes()`.

```
public boolean isLoss (int k, int i)
```

Determines if the agent group `i` is a *loss station* regarding the contact type `k`, i.e., it forwards contacts of type `k` it cannot serve immediately to other agent groups in the system, without queueing them. If the group is not in the ordered list for the contact type or if the group appears at the end of the ordered list, this returns `false`. Otherwise, this returns `true`.

Parameters

`k` the contact type identifier being tested.

`i` the agent group identifier being tested.

Returns `true` if the agent group is a loss station.

Throws

`IndexOutOfBoundsException` if `i` or `k` are negative, `i` is greater than or equal to `Router.getNumAgentGroups()` or `k` is greater than or equal to `Router.getNumContactTypes()`.

```
public boolean isDelay (int k, int i)
```

Determines if the agent group `i` is a *delay station* regarding the contact type `k`, i.e., it queues contacts of type `k` if it cannot serve them immediately. If the group is at the last position in the ordered list for the contact type `k`, this returns `true`. Otherwise, this returns `false`.

Parameters

`k` the contact type identifier being tested.

`i` the agent group identifier being tested.

Returns `true` if the agent group is a delay station.

Throws

`IndexOutOfBoundsException` if `i` or `k` are negative, `i` is greater than or equal to `Router.getNumAgentGroups()` or `k` is greater than or equal to `Router.getNumContactTypes()`.

```
public int getAgentGroupType (int i)
```

Returns the type of the agent group `i` regarding all contact types. This returns `LOSS` if the group is a pure loss station (`isPureLoss (int)` returns `true`), `DELAY` if it is a pure delay station (`isPureDelay (int)` returns `true`), `LOSSDELAY` for a loss/delay station (`isLossDelay (int)` returns `true`), and `NONE` otherwise.

Parameter

`i` the agent group being tested.

Returns the type of the tested agent group.

Throws

`ArrayIndexOutOfBoundsException` if `i` is negative or greater than or equal to `Router.getNumAgentGroups()`.

```
public boolean isPureLoss (int i)
```

Determines if the agent group `i` is a *pure loss station*, i.e., it forwards all contacts to another agent group.

Parameter

`i` the agent group identifier being tested.

Returns `true` if the agent group is a pure loss station, `false` otherwise.

Throws

`ArrayIndexOutOfBoundsException` if `i` is negative or greater than or equal to `Router.getNumAgentGroups()`.

```
public boolean isPureDelay (int i)
```

Determines if the agent group `i` is a *pure delay station*, i.e., it queues all contacts it cannot serve immediately.

Parameter

i the agent group identifier being tested.

Returns `true` if the agent group is a pure delay station, `false` otherwise.

Throws

`ArrayIndexOutOfBoundsException` if *i* is negative or greater than or equal to `Router.getNumAgentGroups()`.

```
public boolean isLossDelay (int i)
```

Determines if the agent group *i* is a *loss/delay station*, i.e., it queues some contacts it cannot serve while forwarding some other contacts to other agent groups.

Parameter

i the agent group identifier being tested.

Returns `true` if the agent group is a loss/delay station, `false` otherwise.

Throws

`ArrayIndexOutOfBoundsException` if *i* is negative or greater than or equal to `Router.getNumAgentGroups()`.

```
public String formatTypeToGroupMap()
```

Calls `RoutingTableUtils.formatTypeToGroupMap (int[][])` with the type-to-group map associated with this router.

Returns the type-to-group map, formatted as a string.

LongestQueueFirstRouter

This extends the queue priority router to select contacts in the longest waiting queue. When a contact arrives into the router, the same scheme for agent group selection as with the queue priority router is used. However, when an agent becomes free, instead of using the group-to-type map to determine the order in which the queues are queried, all queues authorized by the group-to-type map are considered, and a contact is removed from the longest one. If more than one queue has the same maximal size, the contact is removed from the first queue in the ordered list given by the group-to-type map.

```
package umontreal.iro.lecuyer.contactcenters.router;
```

```
public class LongestQueueFirstRouter extends QueuePriorityRouter
```

Constructor

```
public LongestQueueFirstRouter (int[] [] typeToGroupMap, int[] []  
                                groupToTypeMap)
```

Constructs a new longest-queue-first router with a type-to-group map `typeToGroupMap` and a group-to-type map `groupToTypeMap`.

Parameters

`typeToGroupMap` the type-to-group map.

`groupToTypeMap` the group-to-type map.

SingleFIFOQueueRouter

This extends the queue priority router to implement a single FIFO queue. The router assumes that every attached waiting queue uses a FIFO discipline. When a contact arrives into the router, the same scheme for agent group selection as with the queue priority router is used. However, when an agent becomes free, instead of using the group-to-type map to determine the order in which the queues are queried, all queues authorized by the group-to-type map are considered, and the contact with the longest waiting time is removed. If more than one queue has a first contact with the same queue time, which rarely happens in practice, the contact is removed from the first one in the ordered list obtained from the group-to-type map. This policy is equivalent to but more efficient than merging all waiting queues, sorting the contacts in ascending arrival times, and having the agents take the first contact they can serve.

```
package umontreal.iro.lecuyer.contactcenters.router;
```

```
public class SingleFIFOQueueRouter extends QueuePriorityRouter
```

Constructor

```
public SingleFIFOQueueRouter (int[] [] typeToGroupMap, int[] []  
                             groupToTypeMap)
```

Constructs a new single FIFO queue router with a type-to-group map `typeToGroupMap` and a group-to-type map `groupToTypeMap`.

Parameters

`typeToGroupMap` the type-to-group map.

`groupToTypeMap` the group-to-type map.

LongestWeightedWaitingTimeRouter

This extends the queue priority router to select contacts with the longest weighted waiting time. The router assumes that every attached waiting queue uses a FIFO discipline. When a contact arrives into the router, the same scheme for agent group selection as with the queue priority router is used. However, when an agent becomes free, instead of using the group-to-type map to determine the order in which the queues are queried, all queues authorized by the group-to-type map are considered, and the contact with the longest weighted waiting time is removed. More specifically, let w_q be a user-defined weight associated with waiting queue q , and let W_q be the waiting time of the first contact waiting in queue q (if queue q is empty, let $W_q = -\infty$). The router then selects the first contact in the queue with the maximal $w_q W_q$ value. If more than one queue authorized by the freed agent has a first contact with the same weighted waiting time, which rarely happens in practice, the contact is removed from the first queue in the ordered list obtained from the group-to-type map.

```
package umontreal.iro.lecuyer.contactcenters.router;
```

```
public class LongestWeightedWaitingTimeRouter extends QueuePriorityRouter
```

Constructor

```
public LongestWeightedWaitingTimeRouter (int[] [] typeToGroupMap, int[] []
                                         groupToTypeMap, double[]
                                         queueWeights)
```

Constructs a new longest weighted waiting time router with a type-to-group map `typeToGroupMap`, a group-to-type map `groupToTypeMap`, and an array of weights `queueWeights`. Each element of the last array corresponds to a weight assigned to a waiting queue.

Parameters

`typeToGroupMap` the type-to-group map.

`groupToTypeMap` the group-to-type map.

`queueWeights` the array of weights w_q for waiting queues.

Methods

```
public double[] getQueueWeights()
```

Returns the weights associated with each waiting queue. Element q of the returned array contains the weight w_q for the waiting queue q .

Returns the array of weights.

```
public void setQueueWeights (double[] queueWeights)
```

Sets the weights of waiting queues to `queueWeights`.

Parameter

`queueWeights` the new array of weights.

AgentsPrefRouter

Performs agent and contact selection based on user-defined priorities. By default, this router selects the agent with the longest idle time when several agents share the same priority, and the longest waiting time to perform a selection among contacts sharing the same priority. The agents' preference-based router is a generalization of the router taken from [17], using matrices of ranks to take its decisions. The router applies static routing when the ranks are different and uses a dynamic policy when they are equal. This permits the user to partially define the priorities instead of assigning all of them as with the queue priority routing. For example, the user can set the router for the first waiting queue to have precedence over the others while the other queues share the same priority.

Data structures. Two matrices of ranks can be defined, one specifying how contacts prefer agents, and a second one defining how agents prefer contacts. The former matrix, used for agent selection, defines a function $r_{TG}(k, i)$ giving the rank for contacts of type k served by agents in group i . The latter matrix, used for contact selection, defines a function $r_{GT}(i, k)$ giving the rank of contacts of type k when agents in group i perform contact selection. In many cases, one can specify $r_{GT}(i, k)$ only, and have $r_{TG}(k, i) = r_{GT}(i, k)$.

Additionally, the router uses matrices of weights to adjust the priority for candidates with the same rank. These matrices define functions $w_{TG}(k, i)$ and $w_{GT}(i, k)$ which are similar to the ranks functions, except they can take any real number. These matrices are optional and default to matrices of 1's if they are not specified.

Basic routing schemes. The priorities defined by matrices of ranks are used to assign agents to incoming contacts, and contacts to free agents by performing several linear searches over the space of agent groups or waiting queues. Each search constructs or narrows a list of candidates until zero or one candidate is retained. The general algorithm can be summarized as follows.

1. Find a list of candidates sharing the lowest possible rank, or equivalently the highest possible priority;
2. Assign a score to each selected candidate;
3. Select the candidate with the best score.

Agent selection. More specifically, when a new contact of type k arrives, the router constructs an initial list of agent groups for which $N_{F,i}(t) > 0$, and $r_{TG}(k, i) < \infty$. If this list of candidates contains several agent groups, the router compares their ranks $r_{TG}(k, i)$, and retains the agent groups with the minimal rank. If more than one candidates share the same minimal rank, a score is assigned to each of them and the candidate with the best score is taken. The default score of an agent group i is the longest idle time of the agents in that group multiplied by the weight $w_{TG}(k, i)$ (which is 1 by default), also called the longest weighted idle time. For this reason, agent groups linked to this router must be able to take individual agents into account. In the rare event where two candidates share the best score, i.e., two agent groups have the same weighted longest idle time, the candidate

with the smallest index i is retained. If, during the algorithm, the list of candidates happens to be empty, the routed contact is put into a waiting queue corresponding to its type, or blocked if the queue capacity is exceeded. If the list of candidates contains a single agent group, this agent group is selected and service starts.

Note that for a fixed contact type k , if $r_{TG}(k, i)$ is different for all i such that $r_{TG}(k, i) < \infty$, the scheme for agent selection is equivalent to a pure overflow router: each agent group is tested in a fixed order for a free agent. In that setting, the weights $w_{TG}(k, \cdot)$ have no effect. On the other hand, if all finite values of $r_{TG}(k, i)$ for a fixed k are equal, the routing is completely based on the longest-weighted-idle-time selection policy. Any intermediate combination of these two extremes can be achieved by adjusting the ranks appropriately.

Contact selection. Since one waiting queue contains contacts of a single type, we define waiting queue k as the queue containing only contacts of type k . The router assumes that every waiting queue uses a FIFO discipline. When an agent in group i becomes free, an initial list of waiting queues containing at least one contact, and for which $r_{GT}(i, k) < \infty$. If the list of candidates contains several waiting queues, the waiting queues k with the minimal rank are retained. If several waiting queues share this minimal rank, a score is assigned to each candidate, and the waiting queue with the best score is chosen. The default score of a waiting queue k is the weighted waiting time of the first queued contact, i.e., the waiting time multiplied by $w_{GT}(i, k)$. In the rare event where several waiting queues have the same minimal rank, and the same best score, i.e., several queued contacts have the exact same weighted waiting time, the waiting queue with the smallest index k is chosen. If, at any time during the algorithm, the list of candidates becomes empty, the tested agent remains free. When the list of candidates contains a single waiting queue, the first contact in that waiting queue is assigned to the free agent.

Note that for a fixed agent group i , if $r_{GT}(i, k)$ is different for all k such that $r_{GT}(i, k) < \infty$, this policy is equivalent to the queue priority router's contact selection: the waiting queues are queried in a fixed order for contacts. In that particular setting, the weights $w_{GT}(i, \cdot)$ have no effect. On the other hand, if, for a fixed i , all finite $r_{GT}(i, k)$ are equal for all k , the router uses the longest weighted waiting time policy for agent group i . As with agent selection, any combination of these two extremes can be achieved by adjusting the ranks.

Randomized selection. By default, if several agent groups or waiting queues share the same minimal rank, a score is assigned to each of them, and the agent group or queue with the minimal score is chosen. However, this selection can be randomized as follows. Let C_i be the score given to agent group i during agent selection, any negative score excluding the concerned group being replaced with 0. When randomized agent selection is used, the agent group i is selected with probability $p_i = C_i / \sum_{i=0}^{I-1} C_i$. In other words, the highest score an agent group obtains, the greatest is its probability of selection. A similar logic applies for contact selection, with C_i replaced by C_k , the score assigned to contact type k .

```
package umontreal.iro.lecuyer.contactcenters.router;
```

```
public class AgentsPrefRouter extends Router
```

Fields

`protected AgentGroup bestGroup`

Best agent group selected by `selectAgent (Contact, double, boolean[], int)`.

`protected Agent bestAgent`

Best agent selected by `selectAgent (Contact, double, boolean[], int)`, or null if the best agent group does not take account of individual agents.

`protected WaitingQueue bestQueue`

Contains the best waiting queue selected by `selectWaitingQueue (AgentGroup, Agent, double, boolean[], int)`.

`protected DequeueEvent bestQueuedContact`

Contains the best queued contact selected by `selectWaitingQueue (AgentGroup, Agent, double, boolean[], int)`, or null if the first contact in the best queue is taken.

Constructors

`public AgentsPrefRouter (int numTypes, int[][] groupToTypeMap)`

Constructs a new agents' preference-based router with a group-to-type map `groupToTypeMap` and `numTypes` contact types. This router always uses queue priority for contact selection. The rank $r_{GT}(i, k)$ is the value j for which $k_{i,j} = k$ in the group-to-type map, and $r_{TG}(k, i) = r_{GT}(i, k)$. The matrices of weights are initialized with 1's.

Parameters

`numTypes` the number of contact types.

`groupToTypeMap` the group-to-type map.

`public AgentsPrefRouter (double[][] ranksGT)`

Constructs a new agents' preference-based router with matrix of ranks `ranksGT` defining how agents prefer contacts. The given matrix must be rectangular with one row per agent group, and one column per contact type. It defines the function $r_{GT}(i, k)$ while $r_{TG}(k, i) = r_{GT}(i, k)$. The matrices of weights are initialized with 1's.

Parameter

`ranksGT` the contact selection matrix of ranks being used.

Throws

`NullPointerException` if `ranksGT` is null.

`IllegalArgumentException` if the ranks 2D array is not rectangular.

```
public AgentsPrefRouter (double[] [] ranksTG, double[] [] ranksGT)
```

Constructs a new agents' preference-based router with matrix of ranks `ranksTG` defining how contacts prefer agents, and `ranksGT` defining how agents prefer contacts. The given matrices must be rectangular. The matrices of weights are initialized with 1's.

Parameters

`ranksTG` the matrix of ranks defining how contacts prefer agents.

`ranksGT` the matrix of ranks defining how agents prefer contacts.

Throws

`NullPointerException` if `ranksGT` or `ranksTG` are null.

`IllegalArgumentException` if the ranks 2D arrays are not rectangular.

```
public AgentsPrefRouter (double[] [] ranksTG, double[] [] ranksGT, double[] []
                        weightsTG, double[] [] weightsGT)
```

Constructs a new agents' preference-based router with matrix of ranks `ranksTG` defining how contacts prefer agents, and `ranksGT` defining how agents prefer contacts. The matrices of weights are set to `weightsTG`, and `weightsGT`. The given matrices must be rectangular.

Parameters

`ranksTG` the matrix of ranks defining how contacts prefer agents.

`ranksGT` the matrix of ranks defining how agents prefer contacts.

`weightsTG` the matrix of weights defining $w_{TG}(k, i)$.

`weightsGT` the matrix of weights defining $w_{GT}(i, k)$.

Throws

`NullPointerException` if `ranksGT`, `ranksTG`, `weightsTG`, or `weightsGT` are null.

`IllegalArgumentException` if the 2D arrays are not rectangular.

Methods

```
public double[] [] getRanksTG()
```

Returns the matrix of ranks defining how contacts prefer agents, used for agent selection.

Returns the matrix of ranks defining how contacts prefer agents.

```
public void setRanksTG (double[] [] ranksTG)
```

Sets the matrix of ranks defining how contacts prefer agents to `ranksTG`.

Parameter

`ranksTG` the new agent selection matrix of ranks.

Throws

`NullPointerException` if `ranksTG` is null.

`IllegalArgumentException` if `ranksTG` is not rectangular or has wrong dimensions.

```
public double[] [] getRanksGT()
```

Returns the matrix of ranks defining how agents prefer contacts, used for contact selection.

Returns the matrix of ranks defining how agents prefer contacts.

```
public void setRanksGT (double[] [] ranksGT)
```

Sets the matrix of ranks defining how agents prefer contacts to `ranksGT`.

Parameter

`ranksGT` the new contact selection matrix of ranks.

Throws

`NullPointerException` if `ranksGT` is null.

`IllegalArgumentException` if `ranksGT` is not rectangular or has wrong dimensions.

```
public double[] [] getWeightsTG()
```

Returns the matrix of weights defining $w_{TG}(k, i)$.

Returns the matrix of weights defining $w_{TG}(k, i)$.

```
public void setWeightsTG (double[] [] weightsTG)
```

Sets the matrix of weights defining $w_{TG}(k, i)$ to `weightsTG`.

Parameter

`weightsTG` the new matrix of weights defining $w_{TG}(k, i)$.

Throws

`NullPointerException` if `weightsTG` is null.

`IllegalArgumentException` if `weightsTG` is not rectangular or has wrong dimensions.

```
public double[] [] getWeightsGT()
```

Returns the matrix of weights defining $w_{GT}(i, k)$.

Returns the matrix of weights defining $w_{GT}(i, k)$.

```
public void setWeightsGT (double[] [] weightsGT)
```

Sets the matrix of weights defining $w_{GT}(i, k)$ to `weightsGT`.

Parameter

`weightsGT` the new matrix of weights defining $w_{GT}(i, k)$.

Throws

`NullPointerException` if `weightsGT` is null.

`IllegalArgumentException` if `weightsGT` is not rectangular or has wrong dimensions.

```
public double getRankTG (int k, int i)
```

Returns the rank of contact type $k = k$ for agent group $i = i$, used for agent selection.

Parameters

`k` the contact type identifier.

`i` the agent group identifier.

Returns the rank of the contact type for the agent group.

Throws

`ArrayIndexOutOfBoundsException` if `i` or `k` are negative, `i` is greater than or equal to `Router.getNumAgentGroups()`, or `k` is greater than or equal to `Router.getNumContactTypes()`.

```
public double getRankGT (int i, int k)
```

Returns the rank of contact type $k = k$ for agent group $i = i$, used for contact selection.

Parameters

`i` the agent group identifier.

`k` the contact type identifier.

Returns the rank of the contact type for the agent group.

Throws

`ArrayIndexOutOfBoundsException` if `i` or `k` are negative, `i` is greater than or equal to `Router.getNumAgentGroups()`, or `k` is greater than or equal to `Router.getNumContactTypes()`.

```
public double getWeightTG (int k, int i)
```

Returns the weight of contact type $k = k$ for agent group $i = i$, used for agent selection.

Parameters

`k` the contact type identifier.

`i` the agent group identifier.

Returns the weight of the contact type for the agent group.

Throws

`ArrayIndexOutOfBoundsException` if `i` or `k` are negative, `i` is greater than or equal to `Router.getNumAgentGroups()`, or `k` is greater than or equal to `Router.getNumContactTypes()`.

```
public double getWeightGT (int i, int k)
```

Returns the weight of contact type $k = k$ for agent group $i = i$, used for contact selection.

Parameters

`i` the agent group identifier.

`k` the contact type identifier.

Returns the weight of the contact type for the agent group.

Throws

`ArrayIndexOutOfBoundsException` if `i` or `k` are negative, `i` is greater than or equal to `Router.getNumAgentGroups()`, or `k` is greater than or equal to `Router.getNumContactTypes()`.

```
public int[] [] getTypeToGroupMap()
```

Computes a type-to-group map from the agent selection matrix of ranks by calling `RoutingTableUtils.getTypeToGroupMap (double[] [])` on the transpose of the matrix of ranks, and returns the result.

Returns the computed type-to-group map.

```
public int[] [] getGroupToTypeMap()
```

Computes a group-to-type map from the contact selection matrix of ranks by calling `RoutingTableUtils.getGroupToTypeMap (double[] [])`, and returns the result.

Returns the computed group-to-type map.

```
public RandomStream getStreamAgentSelection()
```

Returns the random stream used for agent selection. If the agent selection is not randomized (the default), this returns `null`.

Returns the random stream for agent selection.

```
public void setStreamAgentSelection (RandomStream streamAgentSelection)
```

Sets the random stream for agent selection to `streamAgentSelection`. Setting the stream to `null` disables randomized agent selection.

Parameter

`streamAgentSelection` the new random stream for agent selection.

```
public RandomStream getStreamContactSelection()
```

Returns the random stream used for contact selection. If the contact selection is not randomized (the default), this returns `null`.

Returns the random stream for contact selection.

```
public void setStreamContactSelection (RandomStream streamContactSelection)
```

Sets the random stream for contact selection to `streamContactSelection`. Setting the stream to `null` disables randomized contact selection.

Parameter

`streamContactSelection` the new random stream for contact selection.

```
public AgentSelectionScore getAgentSelectionScore()
```

Returns the current mode of computation for the agent selection score. The default value is `AgentSelectionScore.LONGESTIDLETIME`.

Returns the way the score is computed for agent selection.

```
public void setAgentSelectionScore (AgentSelectionScore
                                   agentSelectionScore)
```

Sets the way scores for agent selection are computed to `agentSelectionScore`.

Parameter

`agentSelectionScore` the way scores for agent selection are computed.

Throws

`NullPointerException` if `agentSelectionScore` is `null`.

```
public ContactSelectionScore getContactSelectionScore()
```

Returns the current mode of computation for the contact selection score. The default value is `ContactSelectionScore.LONGESTWAITINGTIME`.

Returns the way the score is computed for contact selection.

```
public void setContactSelectionScore (ContactSelectionScore
                                     contactSelectionScore)
```

Sets the way scores for contact selection are computed to `contactSelectionScore`.

Parameter

`contactSelectionScore` the way scores for contact selection are computed.

Throws

`NullPointerException` if `contactSelectionScore` is `null`.

```
protected double getRankForAgentSelection (int k, int i)
```

Determines the rank to be used for agent selection for contact type `k`, and agent in group `i`. By default, this returns $r_{TG}(k, i)$, but subclasses may override this method for the rank to depend on some state of the system.

Parameters

k the contact type index.

i the agent group index.

Returns the rank associated with (k, i) .

```
protected void selectAgent (Contact ct, double bestRank, boolean[]
                           candidates1, int numCandidates)
```

This method is called by `selectAgent (Contact)` to perform the selection of an agent among the `numCandidates` agent groups sharing the same minimal finite rank and containing at least one free agent. The method must select an agent group `i` among the agent groups for which `candidates[i]` is `true` and store the reference to this group in the protected field `bestGroup`. The field `bestAgent` can be used to hold the selected agent if the agent group takes account of individual agents. If no candidate is satisfactory, the method must set `bestGroup` and `bestAgent` to `null` before returning; the incoming contact will be queued or blocked as appropriate.

The default implementation computes a score for each candidate using `getScoreForAgentSelection (Contact, AgentGroup, Agent)` and takes the candidate with the best score. This method can be overridden to implement a different selection scheme, e.g., randomly selecting a free agent.

Parameters

ct the contact being processed.

bestRank the best rank found when looking for a candidate agent.

candidates1 the agent group that could serve the contact.

numCandidates the number of `true` values in `candidates`.

```
protected double getScoreForAgentSelection (Contact ct, AgentGroup
                                           testGroup, Agent testAgent)
```

Returns the score for contact `ct` associated with agent group `testGroup` and agent `testAgent`. When selecting an agent for contact `ct`, if there are several agent groups with the same minimal rank, the agent group with the greatest score is selected. Returning a negative infinite score prevents an agent group from being selected. The default `selectAgent (Contact, double, boolean[], int)` method calls this method with `testAgent = null` if `testGroup` is not an instance of `DetailedAgentGroup`, otherwise the method is called with `testAgent = testGroup.getLongestIdleAgent()`.

By default, this returns a score depending on the return value of `getAgentSelectionScore()`. This can return the longest weighted idle time (the default), the weighted number of free agents, or the weight only. See `AgentSelectionScore` for more information.

Parameters

ct the contact being assigned an agent.

testGroup the tested agent group.

testAgent the tested agent, can be `null`.

Returns the score given to the association between the contact and the agent.

`protected double getRankForContactSelection (int i, int k)`

Determines the rank to be used for contact selection for contact type `k`, when an agent in group `i` becomes free. By default, this returns $r_{GT}(i, k)$, but subclasses may override this method for the rank to depend on some state of the system.

Parameters

`i` the agent group index.

`k` the contact type index.

Returns the rank associated with (i, k) .

`protected void selectWaitingQueue (AgentGroup group, Agent agent, double bestRank, boolean[] qCandidates1, int numCandidates)`

Selects the queued contact for an agent `agent` in group `group`, with waiting queue candidates `qCandidates`. The waiting queues `k` for which `qCandidates[k]` is `true` share the same minimal rank and contain at least one contact. The method must store the selected waiting queue in the field `bestQueue` and possibly the contact to be removed in the `bestQueuedContact`. If `bestQueuedContact` is `null`, the first contact in the best queue will be used. If no waiting queue can be selected, `bestQueue` must be set to `null`.

The default implementation selects the waiting queue with the greatest score as computed by `getScoreForContactSelection (AgentGroup, DequeueEvent)`, and the best queued contact is always `null`. This method can be overridden to use an alternate selection scheme, e.g., randomly selecting a queued contact.

Parameters

`group` the agent group containing the free agent.

`agent` the free agent.

`bestRank` the best rank found when searching for candidates.

`qCandidates1` the waiting queue candidates contacts can be pulled from.

`protected double getScoreForContactSelection (AgentGroup group, DequeueEvent ev)`

Returns the score for the association between the agent group `group` and the queued contact represented by `ev`. If contacts can be pulled from several waiting queues with the same minimal rank, the router takes the contact with the greatest score. A negative infinite score prevents a contact from being dequeued.

By default, this returns a score depending on the return value of `getContactSelectionScore()`. This can return the weighted waiting time (the default), the weighted number of queued agents, or the weight only. See `ContactSelectionScore` for more information.

Parameters

`group` the agent group to which pulled contacts would be assigned.

`ev` the dequeue event.

Returns the assigned score.

AgentsPrefRouterWithDelays

Extends the agents' preference-based router to support delays for routing, and allow priority to change with waiting time. Often, a contact has to wait for some time before it can overflow to groups of backup agents. Delays are used to favor the usage of primary agents as opposed to backup agents which are kept for customers which have waited long enough. The priority of a waiting contact may also change if it is waiting long enough. This router allows the user to input such delays, and to set up several different matrices of ranks for priority to be a piecewise-constant function of the waiting time.

Data structures. This router uses the same structures as the agents' preference-based router without delays, with an additional $I \times K$ matrix of delays, and optional extra group-to-type matrices of ranks associated with minimal waiting times. Each delay $d(i, k)$ is a finite positive number indicating the minimal time a contact of type k must wait to be accepted for service by an agent in group i .

Each extra matrix of ranks defines a function $r_{\text{GT},j}(i, k)$ which associates a matrix of ranks with the minimal waiting time w_j . Let $w_0 = 0$ and $r_{\text{GT},0}(i, k) = r_{\text{GT}}(i, k)$. So if no extra matrix of ranks is given, we have only $r_{\text{GT},0}(i, k)$, the default matrix of ranks used by the agents' preference-based routing policy without delays.

Note that fixing $d(i, k) = 0$ for all i and k , and omitting extra matrices of ranks reverts to the original agents' preference-based routing without delays.

Basic routing scheme. We now describe more specifically how the routing with delays works. Let $d_{k,1}, d_{k,2}, \dots$ be the delays $d(\cdot, k)$ sorted in increasing order, with duplicates eliminated, and $d_{k,0} = 0$. When a contact of type k arrives, it can be served only by agents whose group i satisfies $d(i, k) = 0$ in addition to the conditions imposed by the agents' preference-based routing policy. If a contact is queued as no free agent is available to serve it, an event is scheduled to try routing the contact again after a delay $d_{k,1}$. During this so-called rerouting, the delay condition becomes $W \geq d(i, k)$, where W is the time the contact has waited in queue so far. If this second agent selection fails, a third trial happens after a delay $d_{k,2} - d_{k,1}$. More generally, reroutings happen for each delay $d_{k,j}$, for $j = 1, 2, \dots$, unless the contact is accepted by an agent, or abandons. Consequently, as its waiting time increases, the contact can be accepted by a wider range of agents.

Contact selection is done in a similar way as with the agents' preference-based routing policy, except that delays $d(i, k)$, and extra matrices of ranks $r_{\text{GT},j}(i, k)$ are taken into account while determining the rank for a pair (i, k) . More specifically, let W_k be the longest waiting time among all queued contacts of type k . First, the rank of a queued contact of type k is infinite (so the call cannot leave the queue) if its waiting time W_k is smaller than the delay $d(i, k)$. On the other hand, if $W_k \geq d(i, k)$, the rank is given by $r_{\text{GT},j'}(i, k)$ where $j' = \max\{j : W_k \geq w_j\}$ is the index of the matrix of ranks applying to the queued contact. If $j' > 0$, we check the other queued contacts of type k to determine if another queued contact has a smaller rank, i.e., an higher priority. For each scanned queued contact, we check the delay condition and stop scanning as soon as $W_k < d(i, k)$ or $j' = 0$.

The default behavior of this policy can be altered by two switches: overflow transfer, and longest waiting time modes. When overflow transfer is turned ON, a contact gaining access

to some agent groups after waiting some delay also loses access to the original agent groups. When longest waiting time is turned OFF, the contact selection gives priority to pairs (i, k) with small delays $d(i, k)$.

Overflow transfer mode. In this mode, turned off by default, the delay condition for the j th rerouting ($j+1$ th agent selection) becomes $d_{k,j} \leq q < d_{k,j+1}$, j starting with 0, while the original condition is $d_{k,j} \leq q$. With this variant, when a contact has waited sufficient long to overflow to a new set of agent groups, it cannot be served by the original agent groups. Overflow can then be considered as a transfer in a new section of the contact center.

Longest waiting time mode. In this mode, turned on by default, contact selection is performed in a single pass, in a way similar to the contact selection of the policy without delays. However, the delay condition is enforced to restrict contact-to-agent assignment.

If this option is disabled, contact selection is performed using the following multiple-passes process. When an agent in group i becomes free, it first searches for a contact whose type k satisfies $d(i, k) = 0$. Then, it searches for contacts for which $d(i, k) \leq d_{k,1}$, for contacts for which $d(i, k) \leq d_{k,2}$, etc., in that order. This gives higher priority to contacts with small minimal delay, because they can be served by a more restricted set of agents.

The latter behavior of this router is especially appropriate if delays are functions of the distance between the contact and the agent. For local contacts, $d(i, k)$ is small, while it is large for remote contacts. The router then always gives priority to local assignments. However, it is often simpler and more intuitive to use the single-pass contact selection.

```
package umontreal.iro.lecuyer.contactcenters.router;
```

```
public class AgentsPrefRouterWithDelays extends AgentsPrefRouter
```

Constructors

```
public AgentsPrefRouterWithDelays (double[][] ranksGT, double[][] delaysGT)
```

Constructs a new agents' preference-based router with matrix of ranks **ranksGT** and delays matrix **delaysGT**. The given matrices must be rectangular with one row per agent group, and one column per contact type. They define the functions $r_{GT}(i, k)$ (with $r_{TG}(k, i) = r_{GT}(i, k)$), and $d(i, k)$, respectively. The weights matrices are initialized with 1's.

Parameters

ranksGT the contact selection matrix of ranks being used.

delaysGT the delays matrix.

Throws

`NullPointerException` if `ranksGT` or `delaysGT` are null.

`IllegalArgumentException` if the ranks or delays 2D array are not rectangular.

```
public AgentsPrefRouterWithDelays (double[] [] ranksTG, double[] [] ranksGT,
                                   double[] [] delaysGT)
```

Constructs a new agents' preference-based router with matrix of ranks `ranksTG` defining how contacts prefer agents, `ranksGT` defining how agents prefer contacts, and `delaysGT` for routing delays. The given matrices must be rectangular. The weights matrices are initialized with 1's.

Parameters

`ranksTG` the matrix of ranks defining how contacts prefer agents.

`ranksGT` the matrix of ranks defining how agents prefer contacts.

`delaysGT` the delays matrix.

Throws

`NullPointerException` if `ranksGT`, `ranksTG`, or `delaysGT` are null.

`IllegalArgumentException` if the given 2D arrays are not rectangular.

```
public AgentsPrefRouterWithDelays (double[] [] ranksTG, double[] [] ranksGT,
                                   double[] [] weightsTG, double[] []
                                   weightsGT, double[] [] delaysGT)
```

Constructs a new agents' preference-based router with matrix of ranks `ranksTG` defining how contacts prefer agents, and `ranksGT` defining how agents prefer contacts. The weights matrices are set to `weightsTG`, and `weightsGT`. The delays matrix is set to `delaysGT`. The given matrices must be rectangular.

Parameters

`ranksTG` the matrix of ranks defining how contacts prefer agents.

`ranksGT` the matrix of ranks defining how agents prefer contacts.

`weightsTG` the weights matrix defining $w_{TG}(k, i)$.

`weightsGT` the weights matrix defining $w_{GT}(i, k)$.

`delaysGT` the delays matrix.

Throws

`NullPointerException` if `ranksGT`, `ranksTG`, `weightsTG`, `weightsGT`, or `delaysGT` are null.

`IllegalArgumentException` if the 2D arrays are not rectangular.

Methods

```
public double[] [] getDelaysGT()
```

Returns the delays matrix used by this router.

Returns the delays matrix.

```
public double getDelayGT (int i, int k)
```

Returns $d(i, k)$ for the given agent group index i , and contact type identifier k .

Parameters

i the queried agent group index.

k the queried contact type identifier.

Returns the delay $d(i, k)$.

```
public void setDelaysGT (double[] [] delaysGT)
```

Sets the delays matrix of this router to `delaysGT`.

Parameter

`delaysGT` the new delays matrix.

Throws

`NullPointerException` if `delaysGT` is null.

`IllegalArgumentException` if `delaysGT` is not rectangular.

```
public boolean getOverflowTransferStatus()
```

Returns `true` if the overflow transfer mode is enabled. By default, this returns `false`.

Returns the status of the overflow transfer mode.

```
public void setOverflowTransferStatus (boolean overflowTransfer)
```

Sets the overflow transfer mode to `overflowTransfer`.

Parameter

`overflowTransfer` the new status of the mode.

See also `getOverflowTransferStatus()`

```
public boolean getLongestWaitingTimeStatus()
```

Returns `true` if the router uses a single-phase agent selection based on the longest waiting time.

Returns the status of the longest waiting time mode.

```
public void setLongestWaitingTimeStatus (boolean longestWaitingTime)
```

Sets the longest waiting time mode to `longestWaitingTime`.

Parameter

`longestWaitingTime` the new status of the mode.

See also `getLongestWaitingTimeStatus()`

```
public void setRanksGT (double minWaitingTime, double[] [] ranksGT)
```

Sets the matrix of ranks used for selecting contacts which have waited at least `minWaitingTime`.

Parameters

`minWaitingTime` the minimum waiting time.

`ranksGT` the matrix of ranks for this minimal waiting time.

AgentSelectionScore

Parameter indicating how the default agent selection score computed by `AgentsPrefRouter.getScoreForAgentSelection (Contact, AgentGroup, Agent)` is computed.

```
package umontreal.iro.lecuyer.contactcenters.router;  
  
public enum AgentSelectionScore
```

Constants

WEIGHTONLY

The score for an agent group i for contact type k corresponds to the weight $w_{TG}(k, i)$.

NUMFREEAGENTS

The score corresponds to the number of free agents in group i multiplied by the weight $w_{TG}(k, i)$.

LONGESTIDLETIME

The score corresponds to the longest idle time of agents in group i , multiplied by the weight $w_{TG}(k, i)$. Using this score if agent groups are not detailed throws an exception.

ContactSelectionScore

Parameter indicating how the default contact selection score computed by `AgentsPrefRouter.getScoreForContactSelection (AgentGroup, DequeueEvent)` is computed.

```
package umontreal.iro.lecuyer.contactcenters.router;  
  
public enum ContactSelectionScore
```

Constants

WEIGHTONLY

The score for an agent group i for contact type k corresponds to the weight $w_{GT}(i, k)$.

QUEUESIZE

The score corresponds to the number of contacts in queue multiplied by $w_{GT}(i, k)$.

LONGESTWAITINGTIME

The score corresponds to the waiting time of the queued contact of type k multiplied by the weight $w_{GT}(i, k)$.

LocalSpecRouter

This router implements the local-specialist policy which tries to assign contacts to agents in the same region and prefers specialists to preserve generalists. This router associates a region identifier with each contact type and agent group. The *originating region* of a contact is determined by the region identifier associated with its type. The *location* of agents in an agent group is determined by the region identifier associated with the agent group. This policy is similar to agents' preference-based routing, but it adds a region tie breaker and the rank $r_{GT}(i, k)$ can be considered as a measurement of the specialty of agents in group i in serving contacts of type k . Often, $r_{TG}(k, i) = r_{GT}(i, k)$.

When a new contact arrives, the router applies the same agent selection scheme as the agents' preference-based router, except that only agent groups within the originating region of the contact are accepted as candidates. If the contact cannot be served locally, it is added to a waiting queue corresponding to its type. After the contact spent a certain time in queue, called the *overflow delay*, the router tries to perform a new agent selection, this time allowing local and remote agents to serve the contact. If the contact can be served remotely, it is removed from the waiting queue before service starts. Otherwise, it stays in queue.

When an agent becomes free, the same contact selection scheme as with the agents' preference-based router is applied, except that a contact can be pulled from a waiting queue only if its originating region is the same as the location of the free agent. In other words, the local waiting queues are queried first. If, after this first pass, the agent is still free, the router performs a second pass which proceeds the same way as agents' preference-based, except that a contact can be pulled from a waiting queue only if it is in the same region as the free agent, or its waiting time is greater than the overflow delay.

Often, $r_{GT}(i, k) = s(i)$ for each k corresponding to a contact type the agents in group i can serve, and $r_{TG}(k, i) = r_{GT}(i, k)$. The function $s(i)$ is the *skill count* for agent group i , i.e., the number of contact types agents in group i can serve. An agent in group i_1 is more specialist than an agent in group i_2 if $s(i_1) < s(i_2)$. With this format of matrix, if an agent becomes free, local waiting queues are queried first and the contact with the longest weighted waiting time is pulled. Moreover, if weights $w_{GT}(i, k)$ are all set to 1 (the default), only the location of the free agent induces priority for contact selection.

```
package umontreal.iro.lecuyer.contactcenters.router;
```

```
public class LocalSpecRouter extends AgentsPrefRouter
```

Constructors

```
    public LocalSpecRouter (int[] typeRegion, int[] groupRegion, double
                           overflowDelay, int[] skillCounts, boolean[] [] m)
```

Constructs a local-specialist router with contact type region identifiers `typeRegion`, agent group region `groupRegion`, overflow delay `overflowDelay`, skill counts `skillCounts`, and incidence matrix `m`. The rank function $r_{GT}(i, k)$ is set to `skillCounts[i]` if `m[i][k]` is `true` and ∞ otherwise while $r_{TG}(k, i) = r_{GT}(i, k)$. The incidence matrix has one row per agent group and one column per contact type; the boolean `m[i][k]` determines if an agent group `i` can serve a contact type `k`. If `skillCounts` is `null`, the skill count for agent group `i` is determined by counting the number of `k` values for which `m[i][k]` is `true`. The weights matrices are initialized with 1's.

Parameters

`typeRegion` the contact type region identifiers.

`groupRegion` the agent group region identifiers.

`overflowDelay` the delay before overflow is allowed.

`skillCounts` the number of skills for each agent group.

`m` the incidence matrix.

Throws

`NullPointerException` if `typeRegion`, `groupRegion`, or `m` are `null`.

`IllegalArgumentException` if the overflow delay is negative or the incidence matrix is non rectangular.

```
public LocalSpecRouter (int[] typeRegion, int[] groupRegion, double
                        overflowDelay, double[][] ranksGT)
```

Constructs a local-specialist router with contact type region identifiers `typeRegion`, agent group region identifiers `groupRegion`, overflow delay `overflowDelay`, and contact selection ranks matrix `ranksGT`. The agent selection ranks matrix is generated by transposing the contact selection matrix. The weights matrices are initialized with 1's.

Parameters

`typeRegion` the contact type region identifiers.

`groupRegion` the agent group region identifiers.

`overflowDelay` the delay before overflow is allowed.

`ranksGT` the matrix giving the $r_{GT}(i, k)$ function.

Throws

`NullPointerException` if `typeRegion`, `groupRegion`, or `ranks` are `null`.

`IllegalArgumentException` if the overflow delay is negative.

```
public LocalSpecRouter (int[] typeRegion, int[] groupRegion, double
                        overflowDelay, double[][] ranksTG, double[][]
                        ranksGT)
```

Constructs a local-specialist router with contact type region identifiers `typeRegion`, agent group region identifiers `groupRegion`, overflow delay `overflowDelay`, agent selection ranks matrix `ranksTG`, and contact selection ranks matrix `ranksGT`. The weights matrices are initialized with 1's.

Parameters

`typeRegion` the contact type region identifiers.

`groupRegion` the agent group region identifiers.

`overflowDelay` the delay before overflow is allowed.

`ranksTG` the matrix giving the $r_{TG}(k, i)$ function.

`ranksGT` the matrix giving the $r_{GT}(i, k)$ function.

Throws

`NullPointerException` if `typeRegion`, `groupRegion`, or `ranks` are null.

`IllegalArgumentException` if the overflow delay is negative.

```
public LocalSpecRouter (int[] typeRegion, int[] groupRegion, double
                        overflowDelay, double[] [] ranksTG, double[] []
                        ranksGT, double[] [] weightsTG, double[] []
                        weightsGT)
```

Constructs a local-specialist router with contact type region identifiers `typeRegion`, agent group region identifiers `groupRegion`, overflow delay `overflowDelay`, agent selection ranks matrix `ranksTG`, and contact selection ranks matrix `ranksGT`. The weights matrices are set to `weightsTG`, and `weightsGT`.

Parameters

`typeRegion` the contact type region identifiers.

`groupRegion` the agent group region identifiers.

`overflowDelay` the delay before overflow is allowed.

`ranksTG` the matrix giving the $r_{TG}(k, i)$ function.

`ranksGT` the matrix giving the $r_{GT}(i, k)$ function.

`weightsTG` the weights matrix defining $w_{TG}(k, i)$.

`weightsGT` the weights matrix defining $w_{GT}(i, k)$.

Throws

`NullPointerException` if `typeRegion`, `groupRegion`, `ranks` or `weights` matrices are null.

`IllegalArgumentException` if the overflow delay is negative.

Methods

```
public double getOverflowDelay()
```

Returns the current overflow delay for this router.

Returns the current overflow delay.

```
public void setOverflowDelay (double overflowDelay)
```

Sets the overflow delay to `overflowDelay`.

Parameter

`overflowDelay` the new overflow delay.

Throws

`IllegalArgumentException` if the overflow delay is negative.

```
public int getTypeRegion (int k)
```

Returns the region identifier for contact type `k`.

Parameter

`k` the contact type identifier.

Returns the associated region identifier.

Throws

`ArrayIndexOutOfBoundsException` if `k` is negative or greater than or equal to the number of contact types.

```
public void setTypeRegion (int k, int r)
```

Sets the region identifier for contact type `k` to `r`.

Parameters

`k` the contact type identifier.

`r` the new region identifier.

Throws

`ArrayIndexOutOfBoundsException` if `k` is negative or greater than or equal to the number of contact types.

```
public int getGroupRegion (int i)
```

Returns the region identifier for agent group `i`.

Parameter

`i` the agent group identifier.

Returns the associated region identifier.

Throws

`ArrayIndexOutOfBoundsException` if `i` is negative or greater than or equal to the number of agent groups.

```
public void setGroupRegion (int i, int r)
```

Sets the region identifier for agent group `i` to `r`.

Parameters

- i** the agent group identifier.
- r** the new region identifier.

Throws

ArrayIndexOutOfBoundsException if **i** is negative or greater than or equal to the number of agent groups.

```
public int[] [] getTypeToGroupMap()
```

Computes a type-to-group map from the ranks matrix by calling `RoutingTableUtils.getTypeToGroupMap (double[] [], int[], int[])`, and returns the result.

Returns the computed type-to-group map.

```
public int[] [] getGroupToTypeMap()
```

Computes a group-to-type map from the ranks matrix by calling `RoutingTableUtils.getGroupToTypeMap (double[] [], int[], int[])`, and returns the result.

Returns the computed group-to-type map.

QueueRatioOverflowRouter

This router sends new contacts to agent groups using a fixed list, but for each agent group, routing occurs conditional on the expected waiting time. More specifically, the router uses a $K \times I$ ranks matrix giving a rank $r_{\text{TG}}(k, i)$ for each contact type k and agent group i . The lower is this rank, the higher is the priority of assigning contacts of type k to agents in group i . If a rank is ∞ , the corresponding assignment is not allowed. The ranks matrix giving $r_{\text{TG}}(k, i)$ for all k and i is used to generate *overflow lists* defined as follows. For each contact type k , the router creates a list of agent *groupsets* sharing the same priority. The j th groupset for contact type k is denoted $i(k, j) = \{i = 0, \dots, I - 1 \mid r_{\text{TG}}(k, i) = r_{k,j}\}$. Here, $r_{k,j_1} < r_{k,j_2} < \infty$ for any $j_1 < j_2$. The overflow list for contact of type k is then $i(k, 0), i(k, 1), \dots$. For example, suppose we have the following ranks matrix:

$$\begin{pmatrix} 1 & 1 & 2 \\ 1 & \infty & 2 \\ \infty & 1 & \infty \end{pmatrix}$$

The overflow list for contact type 0 is $((0, 1), (2))$, while the overflow list for contact type 1 is $((0), (2))$.

When a new contact of type k arrives, the router performs two phases to assign an agent group or waiting queues to the contact. Here, each waiting queue corresponds to a single agent group. The first phase tries to associate an agent groupset with the contact while the second phase, which occurs when the first phase fails, associates a waiting queue to the contact. The first phase checks every agent groupset $i(k, j)$ sequentially, and stops as soon as a groupset containing a free agent is found. For each considered groupset, the router tests every agent group to determine if at least one agent is free. If a single agent is free, the contact is routed to that agent. If several agents of that groupset are free, the contact is routed to the agent with the longest idle time.

If no agent is available in the tested groupset, one or more waiting queues must be selected to add the contact to. A waiting queue i with size $Q_i(t)$ is associated with a single agent group, which has $N_i(t)$ agents, where t is the current simulation time. The router considers waiting queues in the current groupset only, and selects a queue only if the queue ratio $(S_i(t) + 1)/N_i(t)$ is greater than the agent-group specific target. This queue ratio gives an estimate of the expected waiting time of the contact. Note that this estimate assumes that service times are exponential, and no abandonment is allowed. If no candidate waiting queue is available, e.g., all waiting queues in the groupset have a queue ratio greater than the target queue ratio, the router checks the next agent groupset.

If there are no more groupset, the router performs the second phase as follows. Since no groupset contains a free agent or waiting queue with a small enough queue ratio, the router checks every authorized waiting queue, i.e., each queue i for which $r_{\text{TG}}(k, i) < \infty$, and selects the waiting queue with the smallest queue ratio. In this phase, the queue ratio is allowed to be greater than the target.

The queues the contact is sent to depend on two flags associated with this router: the copy and overflow modes. The copy mode determines if contacts can be queued to multiple

agent groups. The overflow mode, which is used only when contacts can be added into multiple queues, can be set to transfer or promotion. In *transfer* mode, the contact moves from groupsets to groupsets. In *promotion* mode, a copy of the contact is left in every considered groupset.

More specifically, if queueing to multiple targets is disabled, the router always sends the contact to the queue with the smallest queue ratio among the considered candidates. In the first phase, these candidates are the queues in the current groupset with a queue ratio smaller than the target. In the second phase, this corresponds to all queues the contact is authorized in.

If contacts can be added to multiple queues, the overflow mode has the following effect. If candidates were found during the first phase, when checking agent groupset $i(k, j)$, the contact is queued to all queues in that groupset when the overflow mode is transfer. However, if the overflow mode is promotion, the contact is also added to all queues in the preceding groupsets, i.e., groupsets $i(k, j')$ for $j' = 0, \dots, j$. If the router reaches the second phase, the contact is always sent to the queue with the smallest queue ratio. In promotion mode, it is also queued to all other authorized waiting queues.

This router needs agent groups taking individual agents into account to select agents based on their longest idle times.

```
package umontreal.iro.lecuyer.contactcenters.router;

public class QueueRatioOverflowRouter extends Router
```

Constructor

```
public QueueRatioOverflowRouter (int numGroups, double[][] ranksTG, double[]
                                targetQueueRatio, boolean allowCopies,
                                boolean overflowTransfer)
```

Constructs a new queue-ratio overflow router with ranks matrix **ranksTG**, **numGroups** agent groups, and target queue ratio for contact type k set to **targetQueueRatio[k]**. The **allowCopies** flag determines if contacts can be added to multiple waiting queues, while the **overflowTransfer** flag determines if the transfer overflow mode is active. If this latter flag is **false**, overflow mode is set to promotion. The second flag has no effect if the first flag is **false**.

Parameters

numGroups the number of agent groups.

ranksTG the ranks matrix.

targetQueueRatio the target queue ratio for each contact type.

allowCopies the allow-copies flag.

overflowTransfer the overflow-transfer flag.

Throws

`NullPointerException` if any argument is `null`.

`IllegalArgumentException` if the length of `targetQueueRatio` does not correspond to the number of rows in the ranks matrix.

Methods

```
public double[][] getRanksTG()
```

Returns the ranks matrix defining how contacts prefer agents, used for agent selection.

Returns the ranks matrix defining how contacts prefer agents.

```
public void setRanksTG (double[][] ranksTG)
```

Sets the ranks matrix defining how contacts prefer agents to `ranksTG`.

Parameter

`ranksTG` the new agent selection ranks matrix.

Throws

`NullPointerException` if `ranksTG` is `null`.

`IllegalArgumentException` if `ranksTG` is not rectangular or has wrong dimensions.

```
public boolean isAllowCopies()
```

Determines if contacts can be added to multiple queues.

Returns `true` if and only if contacts can be added to multiple queues.

```
public void setAllowCopies (boolean allowCopies)
```

Sets the allow-copies flag to `allowCopies`.

Parameter

`allowCopies` the new value of the flag.

See also `isAllowCopies()`

```
public boolean isOverflowTransfer()
```

Determines if the overflow mode is transfer.

Returns `true` if the overflow mode is transfer.

```
public void setOverflowTransfer (boolean overflowTransfer)
```

Sets the overflow mode to `overflowTransfer`.

Parameter

`overflowTransfer` the new overflow mode.

See also `isOverflowTransfer()`

```
public double[] getTargetQueueRatio()
```

Gets the target queue ratio for each contact type.

Returns the target queue ratios.

```
public void setTargetQueueRatio (double[] targetQueueRatio)
```

Sets the target queue ratio for each contact type to `targetQueueRatio`.

Parameter

`targetQueueRatio` the new target queue ratios.

Throws

`IllegalArgumentException` if the length of `targetQueueRatio` does not correspond to the number of rows in the ranks matrix.

ExpDelayRouter

Represents a router using the expected delay to assign agent groups to new contacts. When a contact is routed to an agent group, it is assigned a free agent of this particular group. If all agents in the target group are busy, the contact enters a waiting queue specific to the target agent group. The contact cannot move across waiting queues.

A waiting queue is associated with each agent group i . When a new contact of type k arrives, the router uses the weighted expected delays $E_i(t)/w_{\text{TG}}(k, i)$ for each waiting queue to take its decisions. Here, $E_i(t)$ is a prediction of the waiting time for the new contact arrived at time t if sent to queue i while $w_{\text{TG}}(k, i)$ is a user-defined constant weight determining the importance of contacts of type k for agents in group i . Two decision modes are available: deterministic, or stochastic. In deterministic mode, the router chooses the agent group with the minimal weighted expected delay. In stochastic mode, the router chooses agent group i with probability

$$p_i(t) = \frac{w_{\text{TG}}(k, i)/E_i(t)}{\sum_{j=0}^{I-1} w_{\text{TG}}(k, j)/E_j(t)}$$

independently of the other contacts. With this formula, the smaller is the weighted expected delay for an agent group i , the higher is the probability of selection of group i . When an agent becomes free, it picks up a new contact from its associated waiting queue only.

Note that the routing of a contact of type k to an agent in group i can be prevented by fixing $w_{\text{TG}}(k, i) = 0$. Increasing $w_{\text{TG}}(k, i)$ increases the probability of a contact of type k to be routed to an agent in group i .

The expected delay is estimated using a waiting time predictor. The default predictor is the `LastWaitingTimePerQueuePredictor` which predicts the waiting time using the last observed waiting time before a service.

```
package umontreal.iro.lecuyer.contactcenters.router;

public class ExpDelayRouter extends Router
```

Constructors

```
public ExpDelayRouter (double[] [] weightsTG, RandomStream stream)
```

Constructs a new router using expected delays, with a weights matrix `weightsTG`, a random stream `stream`. The $K \times I$ weights matrix is used to determine the number of contact types and agent groups while `stream` determines the mode of the router. If `stream` is null, the router is in deterministic mode. Otherwise, it is in stochastic mode.

Parameters

`weightsTG` the weights matrix.

`stream` the random stream used in stochastic mode.

```
public ExpDelayRouter (double[] [] weightsTG, RandomStream stream,
                      WaitingTimePredictor pred)
```

Equivalent to `ExpDelayRouter (double[] [], RandomStream)` with a user-defined waiting time predictor `pred`.

Parameters

`weightsTG` the weights matrix.

`stream` the random stream used in stochastic mode.

`pred` the waiting time predictor.

Methods

```
public double[] [] getWeightsTG()
```

Returns the weights matrix defining $w_{TG}(k, i)$.

Returns the weights matrix defining $w_{TG}(k, i)$.

```
public void setWeightsTG (double[] [] weightsTG)
```

Sets the weights matrix defining $w_{TG}(k, i)$ to `weightsTG`.

Parameter

`weightsTG` the new weights matrix defining $w_{TG}(k, i)$.

Throws

`NullPointerException` if `weightsTG` is null.

`IllegalArgumentException` if `weightsTG` is not rectangular or has wrong dimensions.

```
public RandomStream getStreamAgentSelection()
```

Returns the random stream used for agent selection. If the agent selection is not randomized, this returns null.

Returns the random stream for agent selection.

```
public void setStreamAgentSelection (RandomStream streamAgentSelection)
```

Sets the random stream for agent selection to `streamAgentSelection`. Setting the stream to null disables randomized agent selection.

Parameter

`streamAgentSelection` the new random stream for agent selection.

OverflowAndPriorityRouter

Represents a routing policy allowing contacts to overflow from one set of agents to another, and agents to pick out queued contacts based on priorities that can change at predefined moments during the waiting time. This routing policy also supports some forms of conditional routing. However, the router using this policy might be slow, because of the more complex management of queues. Therefore, if conditional routing is not needed, or if priorities do not change with time, it might be faster to use a simpler policy such as `AgentsPrefRouter` or `AgentsPrefRouterWithDelays`. The latter policy also supports some forms of priorities changing with time.

We now describe the policy in details. The agent selection of any new contact C of type k using this policy is based on a sequence of stages. Each stage is defined by a triplet $(w_{k,j}, f_{k,j}(X, C), g_{k,j}(X, C))$ where $w_{k,j}$ is a minimal waiting time, $f_{k,j}(X, C)$ is a function returning a vector of ranks for agent selection, and $g_{k,j}(X, C)$ is another function returning a vector of ranks for queueing. For any call type $k = 0, \dots, K - 1$, we have $0 \leq w_{k,0} < w_{k,1} < \dots$. Often, we have $f_{k,j} = g_{k,j}$. The vectors returned by these functions can depend on the contact but also on the state X of the system, which allows the implementation of some forms of conditional routing.

More specifically, when a contact of type k arrives, the router checks the first triplet $(w_{k,0}, f_{k,0}, g_{k,0})$. If $w_{k,0} > 0$, the contact waits for $w_{k,0}$ time units in an extra waiting queue no agent has access to; this can be used to model a positive routing delay. Then, the function $f_{k,0}(X, C)$ is evaluated on the new contact C to get a vector of ranks (r_0, \dots, r_{I-1}) . These ranks determine which agent groups can be selected for the new contact, and the priority for each group. The smaller is r_i , the higher is the priority for the agent group i . If $r_i = \infty$, the contact cannot be sent to agent group i at this stage of routing.

The router selects the agent group with the smallest value r_i among the groups containing at least one free agent. If a single group with this minimal rank exists, the contact is sent to a free agent in it, and routing is done. Otherwise, a score S_i is associated with each group with minimal rank, and the group with the highest score is selected. Usually, the score corresponds to the longest idle time of agents in the group.

If no agent group can be assigned to the new contact, the contact is put into one or more waiting queues. There is one priority queue per agent group, and an extra queue storing contacts not queued to any agent group. To select the waiting queues, the router applies the function $g_{k,0}(X, C)$ on the new contact to get a vector (q_0, \dots, q_{I-1}) of ranks. The rank q_i determines the priority of the contact in queue i . The smaller is the rank, the higher is the priority. An infinite rank q_i prevents the contact to be put in queue i . Often, the priority is the same for every waiting queue allowed for the contact, but priorities may differ in general. If all ranks q_i are infinite, the contact goes into the extra queue.

When an agent becomes free, it looks for a contact in the queue associated with its group only. The contacts in this queue are sorted in increasing order of rank. Contacts sharing the same rank are sorted in decreasing order of score. The default function for the score is the time spent in queue. When a contact is removed from a queue, it is also removed from every other queue managed by the router.

If the contact waits for w_1 time units in queue without abandoning or being served, a new agent selection happens. The selection is similar to the first one, except that a new function, $f_{k,1}(X, C)$, is used to generate the vector of ranks. The ranks can thus evolve with time. If no agent group is available for the contact at this second stage of routing, a waiting queue update occurs. For this, a vector of ranks is generated using $g_{k,1}(X, C)$, and used to determine the new priority of the contact, for each queue. If the priority q_i goes from an infinite to a finite value, the contact joins queue i . If the priority goes from a finite to an infinite value, the contact leaves queue i . If the priority changes from a finite value to another finite value, the position of the contact in queue is updated. The priority of a contact can thus evolve with time. This process is repeated at waiting time w_2 , w_3 , and so on, for all stages of routing.

A contact leaving all waiting queues linked to agent groups at a given stage is put into the extra waiting queue. It can still abandon, but it cannot be served until a subsequent stage of routing puts it back into a waiting queue linked to an agent group. On the other hand, if a contact enters a queue linked to an agent group at a given stage of routing, it leaves the extra queue. Moreover, even if the contact changes queue, it keeps the same residual patience time; changing waiting queue does not reset the maximal queue time.

For example, suppose that a contact of type k can be served by two agent groups, 0 and 1. A newly arrived contact has access to group 0 only, and is queued with priority 1 if it cannot be served immediately. However, after s seconds of wait, the contact gains access to group 1. It is queued to this new group with priority 1, but the priority with original group 0 changes to 2 (a lower priority). The parameters for such a routing would be $(0, (1, \infty), (1, \infty)), (s, (1, 1), (2, 1))$. For an example with conditional routing, suppose that at waiting time s , the priorities depend on the service level observed in the last m minutes.

```
package umontreal.iro.lecuyer.contactcenters.router;
```

```
public class OverflowAndPriorityRouter extends Router
```

Constructor

```
public OverflowAndPriorityRouter (int numGroups, RoutingStageInfo[] []
                                stages)
```

Constructs a new overflow and priority router with **numGroups** agent groups, and **stages** for information about routing stages. The 2D array **stages** must contain K rows, each row giving a routing script for a specific contact type.

Parameters

numGroups the number of agent groups.

stages the information about routing stages.

Throws

`NullPointerException` if `stages` is `null`.

`IllegalArgumentException` if `numGroups` is negative or a list of stages are not ordered with respect to waiting time, for at least one contact type.

Methods

```
public double[] [] getWeightsTG()
```

Returns the matrix of weights defining $w_{TG}(k, i)$ for each contact type and agent group. These weights are used by `getScoreForAgentSelection` (`Contact`, `AgentGroup`, `Agent`) to compute scores for agent groups, and default to 1 if they are not set by `setWeightsTG` (`double[] []`).

Returns the matrix of weights defining $w_{TG}(k, i)$.

```
public void setWeightsTG (double[] [] weightsTG)
```

Sets the matrix of weights defining $w_{TG}(k, i)$ for each k and i to `weightsTG`.

Parameter

`weightsTG` the new matrix of weights defining $w_{TG}(k, i)$.

Throws

`NullPointerException` if `weightsTG` is `null`.

`IllegalArgumentException` if `weightsTG` is not rectangular or has wrong dimensions.

```
public double[] [] getWeightsGT()
```

Returns the matrix of weights defining $w_{GT}(i, k)$ for each contact type and agent group. These weights are used by `getScoreForContactSelection` (`DequeueEvent`) to give scores to waiting queues, and default to 1 if they are not set by `setWeightsGT` (`double[] []`).

Returns the matrix of weights defining $w_{GT}(i, k)$.

```
public void setWeightsGT (double[] [] weightsGT)
```

Sets the matrix of weights defining $w_{GT}(i, k)$ for each k and i to `weightsGT`.

Parameter

`weightsGT` the new matrix of weights defining $w_{GT}(i, k)$.

Throws

`NullPointerException` if `weightsGT` is `null`.

`IllegalArgumentException` if `weightsGT` is not rectangular or has wrong dimensions.

```
public AgentSelectionScore getAgentSelectionScore()
```

Returns the current mode of computation for the agent selection score. The default value is `AgentSelectionScore.LONGESTIDLETIME`.

Returns the way the score is computed for agent selection.

```
public void setAgentSelectionScore (AgentSelectionScore  
                                   agentSelectionScore)
```

Sets the way scores for agent selection are computed to `agentSelectionScore`.

Parameter

`agentSelectionScore` the way scores for agent selection are computed.

Throws

`NullPointerException` if `agentSelectionScore` is null.

```
public ContactSelectionScore getContactSelectionScore()
```

Returns the current mode of computation for the contact selection score. The default value is `ContactSelectionScore.LONGESTWAITINGTIME`.

Returns the way the score is computed for contact selection.

```
public void setContactSelectionScore (ContactSelectionScore  
                                     contactSelectionScore)
```

Sets the way scores for contact selection are computed to `contactSelectionScore`.

Parameter

`contactSelectionScore` the way scores for contact selection are computed.

Throws

`NullPointerException` if `contactSelectionScore` is null.

```
protected double getScoreForAgentSelection (Contact ct, AgentGroup  
                                           testGroup, Agent testAgent)
```

Returns the score for contact `ct` associated with agent group `testGroup` and agent `testAgent`. When selecting an agent for contact `ct`, if there are several agent groups with the same minimal rank, the agent group with the greatest score is selected. Returning a negative infinite score prevents an agent group from being selected.

By default, this returns a score depending on the return value of `getAgentSelectionScore()`. This can return the longest weighted idle time (the default), the weighted number of free agents, or the weight only. See `AgentSelectionScore` for more information.

Parameters

`ct` the contact being assigned an agent.

`testGroup` the tested agent group.

`testAgent` the tested agent, can be null.

Returns the score given to the association between the contact and the agent.

```
protected double getScoreForContactSelection (DequeueEvent ev)
```

Returns the score for the queued contact represented by `ev`.

By default, this returns a score depending on the return value of `getContactSelectionScore()`. This can return the weighted waiting time (the default), the weighted number of queued agents, or the weight only. See `ContactSelectionScore` for more information.

Parameter

`ev` the dequeue event.

Returns the assigned score.

Nested class

```
public static final class RoutingInfo
```

Represents information about the routing for a particular contact. When this router processes a contact, it creates an instance of this class and associates it to the contact. This instance can be retrieved by using `contact.getAttributes().get (router)`, where `router` is the corresponding router.

Constructor

```
public RoutingInfo (int numGroups)
```

Constructs a new routing information object for a system with `numGroups` agent groups.

Parameter

`numGroups` the number of agent groups.

Methods

```
public double[] getRanksForAgentSelectionArray()
```

Returns an array containing the ranks associated with this contact for the last agent selection. The first time this method is called, an array is created and returned. This array can then be filled with ranks.

Returns the ranks for agent selection.

```
public double[] getRanksForContactSelectionArray()
```

Similar to `getRanksForAgentSelectionArray()`, for the ranks used by waiting queue selection.

Returns the ranks for waiting queues.

```
public double[] getNewRanksForContactSelectionArray()
```

Similar to `getRanksForAgentSelectionArray()`, for the ranks used by waiting queue selection. This array is used when a new vector of ranks is generated, to be compared with the original vector of ranks giving the priorities of the contacts currently in queues.

Returns the ranks for waiting queues.

```
public DequeueEvent getDequeueEvent (int q)
```

Returns the event representing the associated contact in queue `q`. If the contact is not in this queue, this returns `null`.

Parameter

`q` the index of the tested queue.

Returns the dequeue event.

```
public void setDequeueEvent (int q, DequeueEvent ev)
```

Sets the dequeue event for queue `q` to `ev`.

Parameters

`q` the index of the waiting queue.

`ev` the dequeue event.

```
public int getNumQueues()
```

Returns the number of waiting queues the contact is in.

```
public int getStagesDone()
```

Returns the number of agent selections performed so far for this contact.

```
public void oneStageDone()
```

Indicates that a new agent selection was just done for the contact.

RankFunction

Represents a function computing a vector of ranks for a given contact, for the `OverflowAndPriorityRouter` router.

```
package umontreal.iro.lecuyer.contactcenters.router;  
  
public interface RankFunction
```

Methods

```
public boolean updateRanks (Contact contact, double[] ranks)
```

Fills the array **ranks** with the ranks for the contact **contact**. The given array should have length *I*, and is filled by this method with ranks. The function might use the given contact as well as any relevant model's state to determine the ranks. Note that additional routing information can be obtained through the `OverflowAndPriorityRouter.RoutingInfo`.

The vector of ranks given to this method is constructed by the router, and associated to a specific call. When this method is called for a new call, the vector contains `Double.POSITIVE_INFINITY` values. For any subsequent calls, the vector contains the current ranks for the call. This method should replace these values with the new ranks concerning the call. The method returns **true** if and only if at least one of the ranks in the given vector needs to be updated. Otherwise, it returns **false**.

Parameters

contact the contact being routed.

ranks the vector filled with ranks.

Returns **true** if the vector of ranks was modified, **false** otherwise.

```
public boolean canReturnFiniteRank (int i)
```

Determines if `updateRanks (Contact, double[])` can return a finite rank at position *i* for this particular function.

RoutingStageInfo

Represents a stage for routing, with a minimal waiting time, and two rank functions for agent and contact selections. Instances of this class are used by the `OverflowAndPriorityRouter` router.

```
package umontreal.iro.lecuyer.contactcenters.router;
```

```
public interface RoutingStageInfo
```

Methods

```
public double getWaitingTime()
```

Returns the minimal waiting time for this routing stage.

```
public RankFunction getRankFunctionForAgentSelection()
```

Returns the rank function for agent selection at this stage of routing.

```
public RankFunction getRankFunctionForContactSelection()
```

Returns the rank function for contact selection at this stage of routing.

ExitedContactListener

Represents an exited-contact listener which gets notified when a contact exits the system. A contact can leave the center when it is served, dequeued or blocked.

```
package umontreal.iro.lecuyer.contactcenters.router;  
  
public interface ExitedContactListener
```

Methods

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

This method is called when the contact **contact** is blocked in the router **router**. The integer **bType** is used to indicate the reason of the blocking, e.g., the contact could not be served or put into any waiting queue.

Parameters

router the router causing the blocking.

contact the blocked contact.

bType an indicator giving the reason why the contact is blocked.

```
public void dequeued (Router router, DequeueEvent ev)
```

This method is called when a contact leaves a waiting queue linked to the router **router**, without being served.

Parameters

router the router causing the dequeuing.

ev the dequeue event.

```
public void served (Router router, EndServiceEvent ev)
```

This method is called when a contact was served by an agent. This method is called by the router before the after-contact work begins so **ev** does not contain the information about after-contact time.

Parameters

router the router managing the contact.

ev the end service event.

RoutingTableUtils

Provides some utility methods to manage routing tables represented using 2D arrays. Three types of routing tables are supported: type-to-group and group-to-type maps, incidence matrices and matrices of ranks. This class provides facilities to check the consistency of such routing tables, to generate one table from the other, and to format them as strings. However, converting from one routing table to another may destroy some information or force the conversion algorithm to infer information, which can lead to bad routing policies.

```
package umontreal.iro.lecuyer.contactcenters.router;  
  
public final class RoutingTableUtils
```

Methods

```
public static void checkTypeToGroupMap (int numGroups, int[] []  
                                         typeToGroupMap)
```

Applies a consistency check for the type-to-group map `typeToGroupMap` supporting $I = \text{numGroups}$ agent groups. This checks that every positive element of the given matrix corresponds to an agent group index and no group index appears more than once in an ordered list. If an inconsistency is detected, this throws an `IllegalArgumentException` describing the problem.

Parameters

`numGroups` the number of agent groups I .

`typeToGroupMap` the type-to-group map being checked.

Throws

`NullPointerException` if `typeToGroupMap` or one of its elements are null.

`IllegalArgumentException` if the matrix is incorrect.

```
public static void checkGroupToTypeMap (int numTypes, int[] []  
                                         groupToTypeMap)
```

Applies a consistency check for the group-to-type map `groupToTypeMap` supporting $K = \text{numTypes}$ contact types. This checks that every positive element of the given matrix corresponds to a contact type index and no type index appears more than once in an ordered list. If an inconsistency is detected, this throws an `IllegalArgumentException` describing the problem.

Parameters

`numTypes` the number of contact types K .

`groupToTypeMap` the group-to-type map being checked.

Throws

`NullPointerException` if `groupToTypeMap` or one of its elements are `null`.

`IllegalArgumentException` if the matrix is incorrect.

```
public static void checkConsistency (int[] [] typeToGroupMap, int[] []
                                   groupToTypeMap)
```

Checks the consistency of the routing tables `typeToGroupMap` and `groupToTypeMap`. It is assumed that the matrices are themselves consistent routing tables, i.e., `checkTypeToGroupMap (groupToTypeMap.length, typeToGroupMap)` and `checkGroupToTypeMap (typeToGroupMap.length, groupToTypeMap)` do not throw exceptions. This methods checks that an agent group index i appears in the ordered list for contact type k if and only if the contact type k appears in the ordered list for agent group i . If this consistency criterion is violated, an `IllegalArgumentException` is thrown.

Parameters

`typeToGroupMap` the type-to-group map.

`groupToTypeMap` the group-to-type map.

Throws

`IllegalArgumentException` if the consistency check fails.

See also `checkTypeToGroupMap (int, int[] [])`, `checkGroupToTypeMap (int, int[] [])`

```
public static int[] [] getTypeToGroupMap (int numTypes, int[] []
                                         groupToTypeMap)
```

Generates the type-to-group map from the group-to-type map `groupToTypeMap`. For each contact type k , this method constructs an ordered list containing all agent groups referring to it in the group-to-type map, sorted in increasing order of group identifier. It is assumed that the `groupToTypeMap` matrix is consistent, i.e., `checkGroupToTypeMap (int, int[] [])` does not throw an exception when called with it.

Parameters

`numTypes` the number of contact types K .

`groupToTypeMap` the group-to-type map being processed.

Returns the generated type-to-group map.

See also `checkGroupToTypeMap (int, int[] [])`

```
public static int[] [] getTypeToGroupMap (boolean[] [] m)
```

Constructs and returns a new type-to-group map from the incidence matrix `m`. For each column k of the rectangular matrix m , the method creates a row in the type-to-group map with a column containing value i for each `true` $m(i, k)$ value. This gives lists of agent groups sorted in increasing order of group identifier.

Parameter

`m` the incidence matrix.

Returns the type-to-group map.

See also `ArrayUtil.checkRectangularMatrix (Object)`

```
public static int[] [] getTypeToGroupMap (double[] [] ranksTG)
```

Generates a type-to-group map from the agent selection matrix of ranks `ranksTG`. Assuming that the given matrix is rectangular, the method first uses a scheme similar to `getTypeToGroupMap (boolean[] [])` to get a list of agent groups sorted in increasing order of group identifier for each contact type. Each row of the resulting type-to-group map is then sorted in rank-increasing order, i.e., an agent group i_1 goes before i_2 if $r_{TG}(k, i_1) < r_{TG}(k, i_2)$. If $r_{TG}(k, i_1) = r_{TG}(k, i_2)$, i_1 goes before i_2 in the ordered list for contact type k if $i_1 < i_2$.

Parameter

`ranksTG` the matrix of ranks being transformed.

Returns the generated type-to-group map.

```
public static int[] [] getTypeToGroupMap (double[] [] ranksTG, int[]
                                         typeRegions, int[] groupRegions)
```

This method is similar to `getTypeToGroupMap (double[] [])` with a sorting algorithm adapted for the local-specialist policy. Except from the agent selection matrix of ranks, the method needs arrays associating a region identifier to each contact type and agent group. For contact type k , an agent group i_1 goes before an agent group i_2 if the location of i_1 is the same as the originating region of contacts of type k , but i_2 's location is different from i_1 's. In other words, `groupRegions[i1] == typeRegions[k]` and `groupRegions[i2] != typeRegions[k]` if i_1 is before i_2 . Any pair (i_1, i_2) not meeting this extra condition is sorted using the same algorithm as in `getTypeToGroupMap (double[] [])`.

Parameters

`ranksTG` the matrix of ranks.

`typeRegions` the region identifier of each contact type.

`groupRegions` the region identifier of each agent group.

Returns the constructed type-to-group map.

```
public static int[] [] getGroupToTypeMap (int numGroups, int[] []
                                         typeToGroupMap)
```

Generates the group-to-type map from the type-to-group map `typeToGroupMap`. For each agent group i , this method constructs a list containing all contact types referring to it in the type-to-group map, sorted in increasing order of type identifier. It is assumed that the `typeToGroupMap` matrix is consistent, i.e., `checkTypeToGroupMap (int, int[] [])` does not throw an exception when called with it.

Parameters

numGroups the number of agent groups I .

typeToGroupMap the type-to-group map being processed.

Returns the generated group-to-type map.

See also `checkTypeToGroupMap (int, int[][])`

```
public static int[][] getGroupToTypeMap (boolean[][] m)
```

Constructs and returns a new group-to-type map from the incidence matrix **m**. For each row i of **m**, the method creates a row in the group-to-type map with a column having a value k for each `true` **m**(i , k) value. This gives lists of contact types sorted in increasing order of type identifier.

Parameter

m the incidence matrix.

Returns the group-to-type map.

```
public static int[][] getGroupToTypeMap (double[][] ranksGT)
```

Generates a group-to-type map from the contact selection matrix of ranks **ranksGT**. The method first uses a scheme similar to `getGroupToTypeMap (boolean[][])` to get a list of contact types sorted in increasing order of type identifier for each agent group. Each row of the resulting group-to-type map is then sorted in rank-increasing order, i.e., a contact type k_1 goes before k_2 if $r_{GT}(i, k_1) < r_{GT}(i, k_2)$. If $r_{GT}(i, k_1) = r_{GT}(i, k_2)$, k_1 goes before k_2 in the list if $k_1 < k_2$.

Parameter

ranksGT the matrix of ranks.

Returns the new group-to-type map.

```
public static int[][] getGroupToTypeMap (double[][] ranksGT, int[]
                                         typeRegions, int[] groupRegions)
```

This method is similar to `getGroupToTypeMap (double[][])` with a sorting algorithm adapted for the local-specialist policy. Except from the contact selection matrix of ranks, the method needs arrays associating a region identifier to each contact type and agent group. For each agent group i , a contact type k_1 goes before a contact type k_2 if the originating region of k_1 is the same as the location of agent group i , but k_2 's originating region is different from k_1 's. In other words, `typeRegions[k1] == groupRegions[i]` and `typeRegions[k2] != groupRegions[i]` if k_1 is before k_2 . Any pair (k_1, k_2) not meeting this extra condition is sorted using the same algorithm as in `getGroupToTypeMap (double[][])`.

Parameters

ranksGT the matrix of ranks.

typeRegions the region identifier of each contact type.

groupRegions the region identifier of each agent group.

Returns the constructed group-to-type map.

```
public static boolean[] [] getIncidenceFromTG (int numGroups, int[] []
                                             typeToGroupMap)
```

Constructs and returns the incidence matrix from the `typeToGroupMap` with `numGroups` agent groups. The returned incidence matrix has one row for each agent group and one column for each contact type. Element (i, k) of the matrix is `true` if and only if agent group i is included in the list of contact type k , i.e., $i_{k,j} = k$ for some j . In the incidence matrix, all the ranking induced by the type-to-group map is lost. It is assumed that the type-to-group map is consistent as checked by `checkTypeToGroupMap (int, int[] [])`.

Parameters

`numGroups` the number of agent groups.

`typeToGroupMap` the type-to-group map.

Returns the incidence matrix.

```
public static boolean[] [] getIncidenceFromGT (int numTypes, int[] []
                                             groupToTypeMap)
```

Constructs and returns the incidence matrix from the `groupToTypeMap` with `numTypes` contact types. The returned incidence matrix has one row for each agent group and one column for each contact type. Element (i, k) of the matrix is `true` if and only if the contact type k is included in the list of agent group i , i.e., $k_{i,j} = k$ for some j . In the incidence matrix, all the ranking induced by the group-to-type map is lost. It is assumed that the group-to-type map is consistent as checked by `checkGroupToTypeMap (int, int[] [])`.

Parameters

`numTypes` the number of contact types.

`groupToTypeMap` the group-to-type map.

Returns the incidence matrix.

```
public static double[] [] getRanksFromTG (int numGroups, int[] []
                                          typeToGroupMap)
```

Constructs the agent selection matrix of ranks from the `typeToGroupMap` with `numGroups` agent groups. For each non-negative $i_{k,j} = \text{typeToGroupMap}[k][j]$, the rank $r_{\text{TG}}(k, i_{k,j})$ of contact type k for agent group $i_{k,j}$ is set to j . If i does not appear in the list of k , $r_{\text{TG}}(k, i) = \infty$.

Parameters

`numGroups` the number of agent groups.

`typeToGroupMap` the type-to-group map.

Returns the matrix of ranks.

```
public static double[][] getRanksFromGT (int numTypes, int[][]
                                         groupToTypeMap)
```

Constructs the contact selection matrix of ranks from the `groupToTypeMap` with `numTypes` contact types. For each non-negative $k_{i,j} = \text{groupToTypeMap}[i][j]$, the rank $r_{\text{GT}}(i, k_{i,j})$ of contact type $k_{i,j}$ for agent group i is set to j . If k does not appear in the list of i , $r_{\text{GT}}(k, i) = \infty$.

Parameters

`numTypes` the number of contact types.

`groupToTypeMap` the group-to-type map.

Returns the matrix of ranks.

```
public static double[][] getRanks (boolean[][] m, int[] skillCounts)
```

Constructs a contact selection matrix of ranks from the incidence matrix `m` and skill counts `skillCounts`. Assuming `m` is rectangular, this method creates a matrix of ranks with `m.length` rows and `m[0].length` columns. For each agent group i , and each contact type k , the method sets the rank to ∞ if the contact cannot be served, i.e., if `m[k][i]` is `false`. Otherwise, $r_{\text{GT}}(i, k)$ is set to `skillCounts[i]`. If `skillCounts` is `null`, `skillCounts[i]` is inferred by counting the number of `k` for which `m[i][k]` is `true`.

Parameters

`m` the incidence matrix.

`skillCounts` the skill counts.

Returns the matrix of ranks.

Throws

`IllegalArgumentException` if `m.length` is different from `skillCounts.length`.

```
public static int[][][] getOverflowLists (double[][] ranksTG)
```

Constructs and returns overflow lists from the given matrix of ranks `ranksTG`. More specifically, the ranks matrix giving $r_{\text{TG}}(k, i)$ for all k and i is used to generate *overflow lists* defined as follows. For each contact type k , this method creates a list of agent *groupsets* sharing the same priority. The j th groupset for contact type k is denoted $i(k, j) = \{i = 0, \dots, I - 1 : r_{\text{TG}}(k, i) = r_{k,j}\}$. Here, $r_{k,j_1} < r_{k,j_2} < \infty$ for any $j_1 < j_2$. The overflow list for contact type k is then $i(k, 0), i(k, 1), \dots$. Array `[k][j]` of the returned 3D array contains the elements of $i(k, j)$.

Parameter

`ranksTG` the input matrix of ranks.

Returns the overflow lists.

```
public static String formatTypeToGroupMap (int[] [] typeToGroupMap)
```

Formats the type-to-group ordered lists as a string. For each supported contact type, a line containing **Contact type** *k*: [*i1*, *i2*, ...] is generated, where *i1*, *i2*, ... correspond to agent group indices. Each ordered list is formatted using `formatOrderedList (int[])`.

Parameter

`typeToGroupMap` the type-to-group map being formatted.

Returns the type-to-group map, formatted as a string.

```
public static String formatGroupToTypeMap (int[] [] groupToTypeMap)
```

Formats the group-to-type ordered lists as a string. For each supported agent group, a line containing **Agent group** *i*: [*k1*, *k2*, ...] is generated, where *k1*, *k2*, ... correspond to contact type indices. Each ordered list is formatted using `formatOrderedList (int[])`.

Parameter

`groupToTypeMap` the group-to-type map being formatted.

Returns the group-to-type map, formatted as a string.

```
public static String formatRanksTG (double[] [] ranksTG)
```

Formats the agent selection matrix of ranks **ranksTG** for each contact type and agent group. For each contact type, the returned string contains a line giving the rank of each agent group. When a contact type cannot be served by an agent group, a - is used to represent the infinite rank.

Parameter

`ranksTG` the matrix of ranks to be formatted.

Returns the ranks formatted as a string.

```
public static String formatRanksGT (double[] [] ranksGT)
```

Formats the contact selection matrix of ranks **ranksGT** for each contact type and agent group. For each agent group, the returned string contains a line giving the rank of each contact type. When a contact type cannot be served by an agent group, a - is used to represent the infinite rank.

Parameter

`ranksGT` the matrix of ranks to be formatted.

Returns the ranks formatted as a string.

```
public static String formatWeightsTG (double[] [] weightsTG)
```

Formats the agent selection weights matrix **weightsTG** for each contact type and agent group. For each contact type, the returned string contains a line giving the weight of each agent group. A - is used to represent an infinite weight.

Parameter

`weightsTG` the weights matrix to be formatted.

Returns the weights formatted as a string.

```
public static String formatWeightsGT (double[] [] weightsGT)
```

Formats the contact selection weights matrix `weightsGT` for each contact type and agent group. For each agent group, the returned string contains a line giving the weight of each contact type. A - is used to represent an infinite weight.

Parameter

`weightsGT` the weights matrix to be formatted.

Returns the weights formatted as a string.

```
public static String formatIncidence (boolean[] [] m)
```

Formats the incidence matrix `m` for each contact type and agent group. For each agent group, the returned string contains a line giving the contact types it can serve. Each line contains one value for each contact type. The value 0 is used if the contact cannot be served and 1 otherwise.

Parameter

`m` the incidence matrix to be formatted.

Returns the incidence matrix formatted as a string.

```
public static String formatOrderedList (int[] orderedList)
```

Formats the ordered list `orderedList` as a string. This method constructs and returns a string containing the comma-separated list of indices stored in `orderedList`. If a negative index is found, it is replaced with -1 and formatted in the string only if at least one positive index follows it. For example, the ordered list -2, 0, 3, -1, -1 will be formatted as -1, 0, 3.

Parameter

`orderedList` the ordered list to be formatted.

Returns the string representing the ordered list.

```
public static int[] [] normalizeRoutingTable (int[] [] table, int minColumns)
```

Converts the routing table `table` to a rectangular matrix containing `table.length` rows and at least `minColumns` columns. Assuming the given 2D array is a valid type-to-group or group-to-type map, evaluates the maximum number of columns in each row and pads the rows with -1 for the returned 2D array to be a rectangular matrix, i.e., each row has the same number of columns. In some circumstances, this can simplify manipulation of the routing table and the returned array is still compatible with the routers since negative indices must be ignored.

Parameters

`table` the routing table to be converted.

`minColumns` the minimal number of columns in the normalized routing table.

Returns the converted routing table.

Throws

`NullPointerException` if `table` or one of its elements is `null`.

`IllegalArgumentException` if `minColumns` is negative.

```
public static int[] [] normalizeRoutingTable (int[] [] table)
```

Equivalent to `normalizeRoutingTable (table, 0)`.

AgentGroupSelectors

Provides some convenience methods to select an agent from a list of agent groups. All the methods provided by this class are static and return a reference to the selected agent group. If no agent group is available, they return `null`. They must be given an array of indices `ind` used to reference agent groups in the given router. One can also specify an optional array of booleans `subset` indicating which element in the list will be taken into account.

For each index `j`, let `i = ind[j]`. If `r >= 0` and `subset[j]` is `true` if the subset is specified, the agent group `Router.getAgentGroup(i)` will be considered. Otherwise, `i` will be ignored.

```
package umontreal.iro.lecuyer.contactcenters.router;
```

```
public final class AgentGroupSelectors
```

Methods

```
public static AgentGroup selectFirst (Router router, int[] ind, boolean[]
                                     subset)
```

Selects, from the given ordered list, the first agent group containing at least one free agent.

Parameters

router the router used to map indices in the ordered list to **AgentGroup** references.

ind the ordered list of agent group indices.

subset the subset of indices to take into account when traversing the given list.

Returns the selected agent group.

```
public static AgentGroup selectFirst (Router router, int[] ind)
```

Equivalent to `selectFirst (router, ind, null)`.

```
public static AgentGroup selectLast (Router router, int[] ind, boolean[]
                                     subset)
```

Selects, from the given ordered list, the last agent group containing at least one free agent.

Parameters

router the router used to map indices in the ordered list to **AgentGroup** references.

ind the ordered list of agent group indices.

subset the subset of indices to take into account when traversing the given list.

Returns the selected agent group.

```
public static AgentGroup selectLast (Router router, int[] ind)
    Equivalent to selectLast (router, ind, null).
```

```
public static AgentGroup selectGreatestFree (Router router, int[] ind,
                                             boolean[] subset)
```

Returns a reference to the agent group, among the groups referred to by the given list of indices, containing the greatest number of free agents.

Parameters

router the router used to map indices in the list to **AgentGroup** references.

ind the list of agent group indices.

subset the subset of indices to take into account when traversing the given list.

Returns the selected agent group.

```
public static AgentGroup selectGreatestFree (Router router, int[] ind)
    Equivalent to selectGreatestFree (router, ind, null).
```

```
public static AgentGroup selectUniform (Router router, int[] ind, boolean[]
                                       subset, RandomStream stream)
```

Returns a reference to a randomly selected agent group, among the groups referred to by the given list of indices. The probability of group i to be selected is given by $N_{F,i}(t)/N_F(t)$, where $N_{F,i}(t)$ is the number of free agents in group i at current simulation time, and $N_F(t)$ is the total number of free agents in the groups referred to by the indices.

Parameters

router the router used to map indices in the given list to **AgentGroup** references.

ind the list of agent group indices.

subset the subset of indices to take into account when traversing the given list.

stream the random number stream to generate one uniform.

Returns the selected agent group.

```
public static AgentGroup selectUniform (Router router, int[] ind,
                                       RandomStream stream)
    Equivalent to selectUniform (router, ind, null, stream).
```

```
public static Agent selectLongestIdle (Router router, int[] ind, boolean[]
                                       subset)
```

Returns the reference to the agent having the longest idle time among the agent groups indexed by the list **ind** and possibly restricted by **subset** if it is non-null. This selection rule will be applied only to **DetailedAgentGroup** linked to the router. Indices mapping to an **AgentGroup** instance will be ignored.

Parameters

router the router used to map indices in the given list to **AgentGroup** references.

ind the list of agent group indices.

subset the subset of indices to take into account when traversing the given list.

Returns the selected agent.

```
public static Agent selectLongestIdle (Router router, int[] ind)
```

Equivalent to `selectLongestIdle (router, ind, null)`.

RouterState

Represents state information for a router. This information includes the contents of waiting queues, and the contacts served by agents.

```
package umontreal.iro.lecuyer.contactcenters.router;  
  
public class RouterState
```

Constructor

```
protected RouterState (Router router)  
    Constructs a new state information for a router router.
```

Parameter

router the router being processed.

Methods

```
public AgentGroupState[] getAgentGroups()  
    Returns the state of each agent group saved at the time the state of the router was saved.  
  
Returns the state of agent groups.
```

```
public Map<AgentState, ReroutingState> getAgentReroutingInfo  
()  
    Returns the agent rerouting information saved at the time the state of the router was saved.  
    Each key of the returned map is of type Agent while each value is of type ReroutingState.  
  
Returns the agent rerouting information.
```

```
public Map<DequeEvent, ReroutingState> getContactReroutingInfo  
()  
    Returns the contact rerouting information saved at the time the state of the router was  
    saved. Each key of the returned map is of class DequeEvent while each value is of class  
    ReroutingState.  
  
Returns the contact rerouting information.
```

```
public WaitingQueueState[] getWaitingQueues()  
    Returns the state of the waiting queues attached to the router at the time the state of the  
    router was saved.  
  
Returns the state of the waiting queues.
```

ReroutingState

Represents state information for contact or agent rerouting.

```
package umontreal.iro.lecuyer.contactcenters.router;  
  
public class ReroutingState implements Cloneable
```

Constructor

```
public ReroutingState (int numReroutingsDone, double nextReroutingTime)
```

Constructs a new state information object for rerouting for a contact or an agent that has been previously rerouted `numReroutingsDone` times, and whose next rerouting will happen at time `nextReroutingTime`.

Parameters

`numReroutingsDone` the number of times the contact or agent has been rerouted before.

`nextReroutingTime` the simulation of the next rerouting.

Methods

```
public double getNextReroutingTime()
```

Returns the simulation time at which the router will try to reroute the contact or the agent.

Returns the next rerouting time.

```
public int getNumReroutingsDone()
```

Returns the number of reroutings that has happened so far for the contact or agent.

Returns the number of preceding reroutings.

EnqueueEventWithRerouting

Represents an event that queues a contact, and schedules an additional event for supporting rerouting. This event is the same as the event represented by the superclass `EnqueueEvent`, except that the dequeue event obtained by adding the contact into the waiting queue is used to construct a `ContactReroutingEvent` based on stored information.

```
package umontreal.iro.lecuyer.contactcenters.router;  
  
public class EnqueueEventWithRerouting extends EnqueueEvent
```

Constructors

```
public EnqueueEventWithRerouting (DequeueEvent oldDequeueEvent, Router  
                                targetRouter, ReroutingState  
                                reroutingState)
```

Constructs a new dequeue event with rerouting from the old dequeue event `oldDequeueEvent`, the target router `targetRouter`, and the rerouting state `reroutingState`. This calls `super (oldDequeueEvent)` and sets the target router and rerouting information.

Parameters

`oldDequeueEvent` the old dequeue event.

`targetRouter` the target router.

`reroutingState` the rerouting state.

```
public EnqueueEventWithRerouting (WaitingQueue targetQueue, DequeueEvent  
                                oldDequeueEvent, Router targetRouter,  
                                ReroutingState reroutingState)
```

Constructs a new enqueue event with rerouting from the target waiting queue `targetQueue`, the old dequeue event `oldDequeueEvent`, the target router `targetRouter`, and the rerouting state information `reroutingState`. This calls `super (targetQueue, oldDequeueEvent)` and sets the target router and rerouting information.

Parameters

`targetQueue` the target waiting queue.

`oldDequeueEvent` the old dequeue event.

`targetRouter` the target router.

`reroutingState` the rerouting information.

```
public EnqueueEventWithRerouting (WaitingQueue targetQueue, Contact
                                contact, double queueTime, int dqType,
                                Router targetRouter, int
                                numReroutingsDone, double
                                nextReroutingTime)
```

Constructs a new enqueue event with rerouting from the target waiting queue `targetQueue`, queueing information, and rerouting information. This calls `super (targetQueue, contact, queueTime, dqType)` and sets the target router and rerouting information.

Parameters

`targetQueue` the target waiting queue.

`contact` the contact being queued.

`queueTime` the maximal queue time.

`dqType` the dequeue type.

`targetRouter` the target router.

`numReroutingsDone` the number of times the contact or agent has been rerouted before.

`nextReroutingTime` the simulation of the next rerouting.

Methods

```
public Router getTargetRouter()
```

Returns the target router for this event, i.e., the router for which the rerouting event will be scheduled.

Returns the target router.

```
public double getNextReroutingTime()
```

Returns the simulation time at which the router will try to reroute the contact or the agent.

Returns the next rerouting time.

```
public int getNumReroutingsDone()
```

Returns the number of reroutings that has happened so far for the contact or agent.

Returns the number of preceding reroutings.

Package `umontreal.iro.lecuyer.contactcenters.expdelay`

Provides facilities to predict the waiting time of contacts. This package defines an interface, `WaitingTimePredictor`, representing a waiting-time predictor. Before a predictor can be used, it must be registered with a router. This registration allows the predictor to register any observer necessary to track system's state. A prediction can be obtained for any contact, whether it is newly-arrived, or is already waiting in a queue.

WaitingTimePredictor

Represents a heuristic that can predict the waiting time of a contact depending on the system's state. Such predictions can be used, e.g., for routing, altering patience time, etc. A predictor can have an associated router which is used to obtain system state necessary for predictions. It can also register listeners in order to receive additional information. The method `getWaitingTime (Contact)` is used to get a prediction of the waiting time for a given contact waiting in any queue. The method `getWaitingTime (Contact, WaitingQueue)`, on the other hand, gives a prediction of the waiting time for a contact waiting in a specific queue.

```
package umontreal.iro.lecuyer.contactcenters.expdelay;
```

```
public interface WaitingTimePredictor
```

Methods

```
public Router getRouter()
```

Returns a reference to the router associated with this predictor. By default, this returns `null` since no router is bound to a newly-constructed predictor. A router is associated with a predictor using the `setRouter (Router)` method.

Returns a reference to the currently associated router.

```
public void setRouter (Router router)
```

Sets the router associated with this predictor to `router`. When `router` is non-`null`, this method can also register any listener required to make the predictions. If the router associated with a predictor is changed, the predictor should unregister any listener associated with the previous router.

Parameter

`router` the new router.

```
public void init()
```

Resets any internal variable of this predictor.

```
public double getWaitingTime (Contact contact)
```

Returns a prediction of the waiting time of contact `contact` waiting in any queue. This method returns `Double.NaN` if it cannot make a prediction for the given contact.

Parameter

`contact` the contact for which we need a prediction.

Returns the global waiting time.

```
public double getWaitingTime (Contact contact, WaitingQueue queue)
```

Returns a prediction of the waiting time for the given contact **contact** conditional on the contact joining the waiting queue **queue**. This method returns **Double.NaN** if it cannot make a prediction for the given contact, or the given waiting queue.

Parameters

contact the contact for which a delay is predicted.

queue the target waiting queue.

Returns the predicted delay.

ExpectedDelayPredictor

Approximates the expected waiting time conditional on a given queue assuming that service times are i.i.d. exponential, and a queue is associated with each agent group. More specifically, let $Q_i(t)$ be the size of waiting queue i at time t , and $N_i(t)$ be the total number of agents in group i . We suppose that agents in group i cannot pick up contacts in other queues than queue i . Assuming that service times at agent group i are i.i.d. exponential with mean $1/\mu_i$, the expected waiting time for a contact waiting at queue i is $(Q_i(t) + 1)/(\mu_i N_i(t))$. The rates μ_i are initialized to 1, and should be changed using `setMu (int, double)`. Moreover, if $K = I$, the rates can be initialized automatically using `setMuWithContactTypes()`.

```
package umontreal.iro.lecuyer.contactcenters.expdelay;
```

```
public class ExpectedDelayPredictor implements WaitingTimePredictor
```

Methods

```
public double getMu (int i)
```

Returns the currently used value of μ_i .

Parameter

i the waiting queue index.

Returns the value of μ_i .

```
public double[] getMu()
```

Returns an array containing a copy of the values of μ_i .

Returns an array containing the values of μ_i .

```
public void setMu (int i, double m)
```

Sets the value of μ_i to m .

Parameters

i the index of the affected waiting queue.

m the new value of μ_i .

```
public void setMu (double[] mu)
```

Sets the values of μ_i to mu .

Parameter

mu the array containing the new values of μ_i .

```
public void setMuWithContactTypes()
```

Initializes the values of μ_i using the mean service time for contact types. This method assumes that $K = I$, i.e., there is a waiting queue for each contact type. In that setting, the service rate μ_i is initialized using the mean service time for contact type i .

LastWaitingTimePredictor

Waiting time predictor using the waiting time of the last contact beginning service as a prediction for the waiting time. This predictor monitors every waiting queue attached to the associated router, and stores the last observed waiting time. This waiting time is returned each time a prediction is requested. One can decide if the collected waiting times include times before abandonment, and time before service, using methods `setCollectingAbandonment` (boolean), and `setCollectingService` (boolean), respectively. By default, only the waiting times before service are collected.

```
package umontreal.iro.lecuyer.contactcenters.expdelay;
```

```
public class LastWaitingTimePredictor implements WaitingTimePredictor
```

Methods

```
public boolean isCollectingAbandonment()
```

Determines if the collected waiting times for predictions include times before abandonment. By default, this is set to `false`.

Returns `true` if and only if times of abandonment are used for predicting waiting times.

```
public void setCollectingAbandonment (boolean collectingAbandonment)
```

Sets the flag for collecting abandonment to `collectingAbandonment`.

Parameter

`collectingAbandonment` the new value of the flag.

See also `isCollectingAbandonment()`

```
public boolean isCollectingService()
```

Determines if the collected waiting times for predictions include times before service. By default, this is set to `true`.

Returns `true` if and only if times of beginning of service are used for predicting waiting times.

```
public void setCollectingService (boolean collectingService)
```

Sets the flag for collecting service to `collectingService`.

Parameter

`collectingService` the new value of the flag.

See also `isCollectingService()`

LastWaitingTimePerQueuePredictor

Waiting time predictor using the waiting time of the last contact exiting queue q for service as a prediction for the waiting time of a new contact entering queue q . This predictor collects the waiting times of contacts, and stores the last waiting time separately for each queue. If a prediction is requested for a specific queue, the last waiting time for that queue is given. If a prediction is requested for any queue, the last waiting time over all queues is given.

```
package umontreal.iro.lecuyer.contactcenters.expdelay;

public class LastWaitingTimePerQueuePredictor extends
    LastWaitingTimePredictor
```

HeadOfQueuePredictor

Head of queue waiting time predictor. This predictor obtains a waiting time by taking the longest waiting time among the first queued contacts of the associated router. When waiting queues are first in first out (FIFO), this corresponds to the longest waiting time among all queued contacts. The waiting time of a queued contact is the time from which the contact entered the queue to the current time.

```
package umontreal.iro.lecuyer.contactcenters.expdelay;
```

```
public class HeadOfQueuePredictor implements WaitingTimePredictor
```

Package `umontreal.iro.lecuyer.stat.mperiods`

Provides facilities for storing observations during a simulation per time period, for the common situation where time is partitioned into a finite number of intervals, and statistics have to be collected separately for the different intervals. For example, we may want to collect statistics on the quality of service for each hour in a telephone call center.

Sometimes, a vector or a matrix of statistical probes is sufficient for performing this task, especially when simulating independent replications. Probes are used during an experiment to compute sums and the resulting sums or averages are collected in tallies to get samples; the sample size corresponds to the number of replications. However, the number of periods can sometimes be random. This happens when using the batch means method [13], if the number of batches can change adaptively. For the same reason, it can be necessary to regroup periods to save memory, when the total number of periods becomes too large; this is not supported by a matrix of statistical probes. Events can also be counted in previous batches in addition to the last one. As a result, in a general setting, a probe computing a sum and being reset at the end of each batch cannot be used.

When the number of periods is large, it can be necessary to get observations for a subset of these periods, e.g., the last ten periods. For example, this can be used by the dialer of a phone call center to determine the quality of service in the last ten minutes on which decisions can be based.

To address these problems, this package defines an interface for a matrix of measures. Each row of this matrix corresponds to a type of event, e.g., the type of a customer in a retail store. Each column corresponds to a period that can be any time interval such as half an hour, a complete day, etc.

When independent replications of the same simulation are performed, the finite horizon is often divided into periods. For each period, a vector of observations may be computed and stored. This results in a matrix of observations that can be added to a matrix of tallies at the end of each replication. The matrix can be obtained from the vectors of observations directly, or some vectors can be regrouped.

When simulating for an infinite horizon, a single vector of observations may be obtained. However, to get an estimate on the variance, for computing confidence intervals, the simulation time is divided into intervals called batches. For each batch, a vector of observations is obtained. If the simulation length is constant, or if the number of batches is allowed to be random, it is sufficient to compute one vector at a time and collect it as observations at the end of each batch. However, when the number of batches is required to be constant while the simulation length is random, it is necessary to keep all the vectors of observations to regroup them later. Vectors of observations, or groups of vectors, are collected during or at the end of the simulation.

Depending on the type of experiment, matrices of measures can be added directly to tallies as matrices of observations, some columns can be regrouped, or each column (or group of consecutive columns) might be collected in tallies separately. This package provides a

mechanism to implement simulation events collecting independently of the way the experiment is performed. The **MeasureMatrix** interface represents a matrix of measures. Usually, an implementation of this interface counts the number of occurrences of an event, the sum of values, or some integrals. The **SumMatrix** class implements this interface for computing a matrix of sums. A subclass, **SumMatrixSW**, provides a sliding window permitting the observations in a subset of the periods to be stored. For integrals, the simulator must provide a custom implementation of **MeasureMatrix** which computes a function relative to the simulation time, from time 0 to the current simulation time. The **IntegralMeasureMatrix** class can be used to get the value of the integral for time intervals by storing the value of the integral for user-defined times.

MeasureMatrix

Represents a matrix of measures for a set of related values during successive simulation periods. For example, it can compute the number of served customers of different types, for each simulation period. A period can be any time interval such as half an hour, a complete day, a batch, etc. At the beginning of a simulation, the matrix is initialized using the `init()` method. During the simulation, it is updated with new events or values by implementation-specific methods. An implementation of this interface computes raw observations of a simulated system by counting the number of occurrences of events, by summing values, or by computing integrals. At determined times, e.g., at the end of a replication or a batch, these raw observations are processed to be added into some statistical collectors. This interface provides an abstraction layer to separate the computation of observations from the required processing before they are collected.

Some methods specified by this interface are mandatory whereas others are optional. When an unsupported optional method is called, its implementation simply throws an `UnsupportedOperationException`.

```
package umontreal.iro.lecuyer.stat.mperiods;
```

```
public interface MeasureMatrix
```

Methods

```
public void init()
```

Initializes this matrix of measures for a new simulation replication. This resets the measured values to 0, or initializes the probes used to compute them.

```
public int getNumMeasures()
```

Returns the number of measures calculated by the implementation of this interface.

Returns the number of computed values.

```
public void setNumMeasures (int nm)
```

Sets the number of measures to `nm`. If this method is supported, it can limit the maximal or minimal accepted number of measures.

Parameter

`nm` the new number of measures.

Throws

`IllegalArgumentException` if the given number is negative or not accepted.

`UnsupportedOperationException` if the number of measures cannot be changed.

```
public int getNumPeriods()
```

Returns the number of periods stored into this matrix of measures.

Returns the number of stored periods.

```
public void setNumPeriods (int np)
```

Sets the number of periods of this matrix to `np`. If this method is supported, it can limit the maximal or minimal accepted number of periods.

Parameter

`np` the new number of periods.

Throws

`IllegalArgumentException` if the given number is negative or not accepted.

`UnsupportedOperationException` if the number of periods cannot be changed.

```
public double getMeasure (int i, int p)
```

Returns the measure corresponding to the index `i` and period `p`.

Parameters

`i` the index of the measure.

`p` the period of the measure.

Returns the corresponding value.

Throws

`IndexOutOfBoundsException` if `i` or `p` are negative or greater than or equal to the number of measures or the number of periods, respectively.

```
public void regroupPeriods (int x)
```

Increases the length of stored periods by regrouping them. If this method is supported, for `p = 0, ..., getNumPeriods()/x - 1`, it sums the values for periods `xp, ..., xp+x-1`, and stores the results in period `p` whose length will be `x` times the length of original periods. If the number of periods is not a multiple of `x`, an additional period is used to contain the remaining sums of values. The unused periods are zeroed for future use. This method can be useful for memory management when using batch means to estimate steady-state performance measures.

Parameter

`x` the number of periods per group.

Throws

`IllegalArgumentException` if the number of periods per group is negative or 0.

`UnsupportedOperationException` if the matrix does not support regrouping.

StatProbeMeasureMatrix

Matrix of measures whose value is obtained using a statistical probe. This matrix only contains one measure and one period, and its value is obtained by using `StatProbe.sum()`. Since the sum can be considered as an integral, the `IntegralMeasureMatrix` class can be used to turn this single-period matrix into a multiple-periods one.

```
package umontreal.iro.lecuyer.stat.mperiods;
```

```
public class StatProbeMeasureMatrix implements MeasureMatrix, Cloneable
```

Constructor

```
public StatProbeMeasureMatrix (StatProbe probe)
```

Constructs a new matrix of measures using the statistical probe `probe`.

Parameter

`probe` the statistical probe being used.

Methods

```
public StatProbe getStatProbe()
```

Returns the statistical probe associated with this matrix.

Returns the associated statistical probe.

```
public void setStatProbe (StatProbe probe)
```

Sets the associated statistical probe to `probe`. If `null` is given, this changes the number of measures and periods to 0. If a non-null probe is given, the number of measures in this object is 1.

Parameter

`probe` the new statistical probe.

```
public void setNumMeasures (int nm)
```

Throws an `UnsupportedOperationException`.

Throws

`UnsupportedOperationException` if this method is called.

```
public void setNumPeriods (int np)
```

Throws an `UnsupportedOperationException`.

Throws

`UnsupportedOperationException` if this method is called.

```
public void regroupPeriods (int x)
```

Throws an `UnsupportedOperationException`.

Throws

`UnsupportedOperationException` if this method is called.

```
public StatProbeMeasureMatrix clone()
```

Makes a copy of this matrix of measures. The statistical probe is not cloned.

Returns a clone of this instance.

ListOfStatProbesMeasureMatrix

Matrix of measures whose values are obtained using an list of statistical probes. This matrix contains one measure for each element of the list, and a single period. The measures are obtained by using the `StatProbe.sum()` method. Since the sum can be considered as an integral, the `IntegralMeasureMatrix` can be used to turn this single-period matrix into a multiple-periods one if needed.

```
package umontreal.iro.lecuyer.stat.mperiods;  
  
public class ListOfStatProbesMeasureMatrix implements MeasureMatrix,  
    Cloneable
```

Constructor

```
public ListOfStatProbesMeasureMatrix (ListOfStatProbes<? extends StatProbe  
    > list)
```

Constructs a new matrix of measures using the list of probes `list`.

Parameter

`list` the list of statistical probes being used.

Methods

```
public ListOfStatProbes<? extends StatProbe> getListOfStatProbes  
( )
```

Returns the list of statistical probes associated with this matrix.

Returns the associated list of statistical probes.

```
public void setListOfStatProbes (ListOfStatProbes<? extends StatProbe>  
    list)
```

Sets the associated list of statistical probes to `list`. If the given list is `null`, the number of measures and periods is set to 0. Otherwise, the number of measures corresponds to the length of the list, and the number of periods is 1.

Parameter

`list` the new list of statistical probes.

```
public void setNumPeriods (int np)
```

Throws an `UnsupportedOperationException`.

Throws

UnsupportedOperationException if this method is called.

```
public void regroupPeriods (int x)
```

Throws an UnsupportedOperationException.

Throws

UnsupportedOperationException if this method is called.

```
public ListOfStatProbesMeasureMatrix clone()
```

Makes a copy of this matrix of measures. The list of statistical probes is not cloned.

Returns a clone of this instance.

MatrixOfStatProbesMeasureMatrix

Matrix of measures whose values are obtained using a matrix of statistical probes. This matrix contains one measure for each row of the matrix, and one period for each column. The measures are obtained using the `StatProbe.sum()` method. Since the sum can be considered as an integral, the `IntegralMeasureMatrix` can be used to turn this matrix into a multiple-periods one if the associated matrix contains a single column.

```
package umontreal.iro.lecuyer.stat.mperiods;  
  
public class MatrixOfStatProbesMeasureMatrix implements MeasureMatrix,  
        Cloneable
```

Constructor

```
public MatrixOfStatProbesMeasureMatrix (MatrixOfStatProbes<?> matrix)  
    Constructs a new matrix of measures using the matrix of probes matrix.
```

Parameter

`matrix` the matrix of statistical probes being used.

Methods

```
public MatrixOfStatProbes<?> getMatrixOfStatProbes()  
    Returns the matrix of statistical probes associated with this matrix of measures.
```

Returns the associated matrix of statistical probes.

```
public void setMatrixOfStatProbes (MatrixOfStatProbes<?> matrix)  
    Sets the associated matrix of statistical probes to matrix. If the given matrix is null, the number of measures and periods are set to 0. Otherwise, they correspond to the number of rows and columns of the matrix, respectively.
```

Parameter

`matrix` the new matrix of statistical probes.

```
public void regroupPeriods (int x)  
    Throws an UnsupportedOperationException.
```

Throws

`UnsupportedOperationException` if this method is called.

```
public MatrixOfStatProbesMeasureMatrix clone()  
    Makes a copy of this matrix of measures. The statistical probe matrix is not cloned.
```

Returns a clone of this instance.

MeasureSet

Represents a set of related measures computed using different measure matrices. Each measure of such a set corresponds to a measure computed by another matrix. For example, this class can regroup the queue size for different waiting queues. It can compute the sum of the measures for each period, and give statistical collecting mechanisms access to the measures using the `MeasureMatrix` interface.

```
package umontreal.iro.lecuyer.stat.mperiods;  
  
public class MeasureSet implements MeasureMatrix, Cloneable
```

Constructor

```
public MeasureSet()
```

Constructs a new empty measure set. The `addMeasure (MeasureMatrix, int)` method must be used to add some measures.

Methods

```
public boolean isComputingSumRow()
```

Determines if the measure set contains an additional row containing the sum of each column. If this returns `true` (the default), the row of sums is computed. Otherwise, it is not computed. The sum row adds one additional measure to the measure set only if the number of measures is greater than 1.

Returns the sum row computing indicator.

```
public void setComputingSumRow (boolean b)
```

Sets the computing sum row indicator to `b`. See `isComputingSumRow()` for more information.

Parameter

`b` the new sum row computing indicator.

```
public void addMeasure (MeasureMatrix mat, int imat)
```

Adds the measure `imat` calculated by `mat` to this set of measures. It is recommended that every added measure matrix has the same number of periods.

Parameters

`mat` the measure matrix computing the added measure.

`imat` the index of the added measure, in `mat`.

Throws

`NullPointerException` if `mat` is `null`.

```
public void clearMeasures()
```

Clears all measures contained in this set.

```
public MeasureInfo getMeasureInfo (int i)
```

Returns the measure information object for measure `i`.

Returns the measure information object.

Throws

`IndexOutOfBoundsException` if `i` is out of bounds.

```
public int getNumMeasures()
```

Returns the number of supported measures. If the set contains 0 or 1 measure, this method returns 0 or 1, respectively. If the set contains $n > 1$ measures, $n + 1$ is returned if `isComputingSumRow()` returns `true`, or n is returned otherwise.

Returns the number of supported measures.

```
public int getNumPeriods()
```

Returns the number of supported periods. If the set is empty, this returns 0. Otherwise, this returns the maximal number of periods of the contained measure matrices.

Returns the supported number of periods.

```
public void setNumMeasures (int nm)
```

This implementation does not support changing the number of measures.

Throws

`UnsupportedOperationException` if this method is called.

```
public void setNumPeriods (int np)
```

This implementation does not support changing the number of periods.

Throws

`UnsupportedOperationException` if this method is called.

```
public void regroupPeriods (int x)
```

This implementation does not support period regrouping.

Throws

`UnsupportedOperationException` if this method is called.

```
public void init()
```

This method does nothing in this implementation.

```
public double getMeasure (int i, int p)
```

Returns the measure *i* in period *p* for this matrix. Let *n* be the number of measures in this set, i.e., the value of `getNumMeasures()` if `isComputingSumRow()` returns `false`. If $i < n$, this returns the *i*th measure added to this set. If $i = n$, $n > 1$ and the measure set is computing the sum row, this returns the sum of all the contained measures for period *p*. Let *P* be the number of periods as returned by `getNumPeriods()`. If *p* is greater than or equal to the number of periods in the queried measure matrix but smaller than *P*, `Double.NaN` is returned. In the sum of measures, the NaN value is not counted to avoid a NaN sum.

Parameters

i the index of the measure.

p the index of the period.

Returns the value of the measure.

Throws

`IndexOutOfBoundsException` if the measure or period indices are out of bounds.

Nested class

```
public static final class MeasureInfo implements Cloneable
```

Contains information about a measure added to a measure set.

Constructor

```
public MeasureInfo (MeasureMatrix mat, int index)
```

Constructs a new measure information object for the measure *index* in the matrix *mat*.

Parameters

mat the measure matrix to take the measure from.

index the index of the measure, in *mat*.

Throws

`NullPointerException` if *mat* is null.

Methods

`public MeasureMatrix getMeasureMatrix()`

Returns the measure matrix from which the measure is extracted.

Returns the associated measure matrix.

`public int getMeasureIndex()`

Returns the index, in the associated measure matrix, of the represented measure.

Returns the index of the measure.

SumMatrix

This matrix of measures can be used to compute sums of values, or the number of occurrences of events. It supports several types of observations on several simulation periods. This class supports every optional operation specified by the `MeasureMatrix` interface.

```
package umontreal.iro.lecuyer.stat.mperiods;
```

```
public class SumMatrix implements MeasureMatrix, Cloneable
```

Fields

```
protected int numTypes
```

Number of types of events.

```
protected int numPeriods
```

Number of periods.

```
protected int numStoredPeriods
```

Number of stored periods.

```
protected double[] count
```

Array containing the sums, in row major order. If `i` is an event type and `p` is a stored period, the value at `(i, p)` is given by `count[i + numTypes*p]`.

Constructors

```
public SumMatrix (int numTypes)
```

Constructs a new matrix of sums for `numTypes` event types and a single period.

Parameter

`numTypes` the number of event types.

Throws

`IllegalArgumentException` if the number of types is negative.

```
public SumMatrix (int numTypes, int numPeriods)
```

Constructs a new matrix of sums for `numTypes` event types and `numPeriods` periods.

Parameters

`numTypes` the number of event types.

`numPeriods` the number of stored periods.

Throws

`IllegalArgumentException` if the number of types or periods is negative.

Methods

```
public int getNumStoredPeriods()
```

Returns the total number of periods stored in this matrix of sums. This corresponds to $p+1$ if p is the maximal period index given to `add (int, int, double)` or `set (int, int, double)` since the last call to `init()`. If `add (int, int, double)` or `set (int, int, double)` were not called since the last initialization, this returns 0. The returned value cannot be larger than the number of stored periods (`getNumPeriods()`).

Returns the number of used periods.

```
public void add (int type, int period, double x)
```

Adds a new observation `x` of type `type` in the period `period`.

Parameters

`type` the type of the new value.

`period` the period of the new value.

`x` the value being added.

Throws

`ArrayIndexOutOfBoundsException` if `type` or `period` are negative, if `type` is greater than or equal to the number of supported types, or if `period` is greater than or equal to the number of supported periods.

```
public void add (int type, int period, double x, DoubleDoubleFunction fn)
```

Similar to `add (int, int, double)`, but applies a function `fn` instead of just adding. More specifically, if c is the original value in the matrix, and x is the new value, this method adds the value $f(c, x)$ at the given position in the matrix.

```
public void set (int type, int period, double x)
```

Sets the sum for event `type` in period `period` for this matrix to `x`. This is the same as the `add (int, int, double)` method except the measure is replaced by `x` instead of being incremented.

Parameters

`type` the type of the event.

`period` the period of the event.

`x` the new value.

Throws

`ArrayIndexOutOfBoundsException` if `type` or `period` are negative, if `type` is greater than or equal to the number of supported types, or if `period` is greater than or equal to the number of supported periods.

```
public void setNumMeasures (int nm)
```

Sets the number of measures to `nm`. If `nm` is greater than `getNumMeasures()`, new measures are added and initialized to 0. If `nm` is smaller than `getNumMeasures()`, the last `getNumMeasures() - nm` measures are removed. Otherwise, nothing happens.

Parameter

`nm` the new number of measures.

Throws

`IllegalArgumentException` if the given number is negative.

```
public void setNumPeriods (int np)
```

Sets the number of periods to `np`. As with `setNumMeasures (int)`, added periods are initialized to 0 and the last periods are removed if necessary.

Parameter

`np` the new number of periods.

Throws

`IllegalArgumentException` if the given number is negative.

```
protected void regroupPeriods (int x, boolean onlyFirst)
```

Similar to `regroupPeriods (int)`, but if `onlyFirst` is `false`, do not sum the values in each group. `regroupPeriods` with `onlyFirst = false` is internally used by `IntegralMeasureMatrix`.

```
protected int getPeriod (int p)
```

Returns the period index corresponding to period `p`. This returns `p` by default, but it is overridden by `SumMatrixSW` to take the sliding window into account.

Parameter

`p` the source period index.

Returns the target period index.

SumMatrixSW

Extends **SumMatrix** to add a sliding window. By using a circular buffer to store the values, it can compute observations for all the periods or for a subset of the periods.

When values are added to this matrix of sums using the **add (int, int, double)** method, the index of a *real period* needs to be specified. When obtaining a value from this object, the index of a *stored period* (or simply a period) is needed. If the number of considered periods is smaller than or equal to the number of stored periods, these two index spaces match, no information is lost, and this class behaves exactly as **SumMatrix**. This is the most common case.

However, if the number of real periods is greater than the number of stored periods, some values are lost. Only the values from the last periods are accessible at any time. For example, if a matrix of sums is defined to store 10 periods, values for the periods 0 to 9 are available until the **add (int, int, double)** method is required to add a value in the period 10 or greater. After a value is added into the real period 10, values for the period 0 are lost. The stored periods are shifted and stored periods 0 to 9 then correspond to real periods 1 to 10. This facility allows, for example, to compute a statistic for the last ten minutes, at any times during a simulation.

```
package umontreal.iro.lecuyer.stat.mperiods;

public class SumMatrixSW extends SumMatrix
```

Constructors

```
public SumMatrixSW (int numTypes)
```

Constructs a new matrix of sums with sliding window for **numTypes** event types and a single period.

Parameter

numTypes the number of event types.

Throws

IllegalArgumentException if the number of types is negative.

```
public SumMatrixSW (int numTypes, int numPeriods)
```

Constructs a new matrix of sums with sliding window for **numTypes** event types and **numPeriods** periods.

Parameters

numTypes the number of event types.

numPeriods the number of stored periods.

Throws

`IllegalArgumentException` if the number of types or periods is negative.

Methods

```
public int getFirstRealPeriod()
```

Returns the real period corresponding to stored period having index 0 when using the `SumMatrix.getMeasure (int, int)` method. If no period is currently lost, this returns 0.

Returns the first real period.

```
public void setFirstRealPeriod (int firstRealPeriod)
```

Sets the index of the first real period to `firstRealPeriod`.

Parameter

`firstRealPeriod` the new index of the first real period.

Throws

`IllegalArgumentException` if `firstRealPeriod` is negative.

```
public int getNumRealPeriods()
```

Returns the number of real periods used by this matrix of sums. This corresponds to one plus the maximal value of `realPeriod` given to `add (int, int, double)` or `set (int, int, double)` since the last call to `init()`. In contrast with the similar method `SumMatrix.getNumStoredPeriods()`, the returned value can be greater than `SumMatrix.getNumPeriods()`.

Returns the number of used real periods.

```
public void add (int type, int realPeriod, double x)
```

Adds a new observation `x` of type `type` in the real period `realPeriod`. If the given real period is inside the interval `[getFirstRealPeriod(), ..., getFirstRealPeriod() + SumMatrix.getNumPeriods() - 1]`, which we will call the current interval, a new value is added; the corresponding measure is increased by `x`. If the period is at the left of the current interval, an exception is thrown. Otherwise, the window of observations slides and `getFirstRealPeriod()` is increased; some computed values are then lost.

Parameters

`type` the type of the new value.

`realPeriod` the real period of the new value.

`x` the value being added.

Throws

`ArrayIndexOutOfBoundsException` if `type` or `realPeriod` are negative, if `type` is greater than or equal to the number of supported types, or if `realPeriod` is smaller than `getFirstRealPeriod()`.

```
public void set (int type, int realPeriod, double x)
```

Sets the sum for event `type` in real period `realPeriod` for this matrix to `x`. This is the same as the `add (int, int, double)` method except the measure is replaced by `x` instead of being incremented.

Parameters

`type` the type of the event.

`realPeriod` the real period of the event.

`x` the new value.

Throws

`ArrayIndexOutOfBoundsException` if `type` or `realPeriod` are negative, if `type` is greater than or equal to the number of supported types, or if `realPeriod` is smaller than `getFirstRealPeriod()`.

IntegralMeasureMatrix

Computes per-period values for a matrix of measures with a single period. Some matrices of measures only support a single period. For example, when using an `Accumulate` object to compute an integral over simulation time, per-period measures cannot be computed directly. This class can be used to transform a matrix of measures with a single period computing integrals into a multiple-periods matrix.

Let $f_i(t)$ be an integral (or a sum) for measure i , computed by the underlying single-period matrix over the simulation time, from 0 to t . Often, $f_i(t)$ is a discrete function such as a sum or the integral of a piecewise-constant function, but the function can also be continuous. If `newRecord()` is called at simulation time t_p , the value of $\mathbf{f}(t_p) = (f_0(t_p), f_1(t_p), \dots)$ is computed and recorded. At time t_0 where `init()` is called, a record is automatically added with the values in the matrices of measures. At the end of the simulation, if `newRecord()` was called P times, we have $P + 1$ recorded values of $\mathbf{f}(t_p)$, for $p = 0, \dots, P$. This permits the computation of P vectors of integrals, each corresponding to a period. The integrals for period p , i.e., during the interval $[t_p, t_{p+1})$, are computed by $\mathbf{f}(t_{p+1}) - \mathbf{f}(t_p)$.

```
package umontreal.iro.lecuyer.stat.mperiods;
```

```
public class IntegralMeasureMatrix<M extends MeasureMatrix> implements
    MeasureMatrix, Cloneable
```

Constructor

```
public IntegralMeasureMatrix (M mat, int numPeriods)
```

Constructs a new matrix of measures for computing integrals on multiple periods. The wrapped matrix of measures is given by `mat`, and the integral is computed for `numPeriods`. The object will be able to record `numPeriods+1` values of $\mathbf{f}(t)$. The number of measures or periods of `mat` should not be changed after it is associated with this object.

Parameters

`mat` the single-period matrix of measures.

`numPeriods` the required number of periods.

Throws

`NullPointerException` if `mat` is null.

`IllegalArgumentException` if a multiple-periods matrix of measures is given, or if `numPeriods` is smaller than 1.

Methods

```
protected SumMatrix createSumMatrix (int nm, int np)
```

This methods creates and returns the internal sum matrix, and is overridden in `IntegralMeasureMatrixSW` to create an instance of `SumMatrixSW` instead.

Parameters

nm the number of measures.

np the number of periods.

public M getMeasureMatrix()

Returns the associated single-period matrix of measures.

Returns the associated single-period matrix of measures.

public void setMeasureMatrix (M mat)

Sets the associated matrix of measures to **mat**. This should only be called before or after **init()**.

Parameter

mat the new matrix of measures.

Throws

NullPointerException if **mat** is null.

IllegalArgumentException if the given matrix has multiple periods.

public SumMatrix getSumMatrix()

Returns the internal sum matrix for which each period p contains the value of $\mathbf{f}(t_p)$. The number of measures of this matrix is **getNumMeasures()** while the number of periods is one more than **getNumPeriods()**.

Returns the internal matrix of sums.

public int getNumStoredRecords()

Returns the current number of records of $\mathbf{f}(t)$ available for this matrix of measures.

Returns the current number of records.

public void newRecord()

Records the current values of $\mathbf{f}(t)$. This increases the number of stored records, and an **IllegalStateException** is thrown if no additional record can be stored.

protected int getPeriod()

Returns the period, in **mpc**, the new record needs to be added in. This returns **mpc.getNumStoredPeriods()**.

Returns the period used by **newRecord()**.

public double getSum (int i, int r)

Returns $f_i(t_r)$, the measure **i** of the associated measure matrix at the simulation time t_r .

Parameters

i the measure index.

r the record index.

Returns the value of the measure.

Throws

`IndexOutOfBoundsException` if **i** or **r** are out of bounds.

```
public double getMeasure (int i, int p)
```

Returns the measure **i** for period **p**. This corresponds to $f_i(t_{p+1}) - f_i(t_p)$ where t_p is the simulation time of the stored record p .

Parameters

i the index of the measure.

p the period of the measure.

Returns the corresponding value.

Throws

`IndexOutOfBoundsException` if **i** or **p** are negative or greater than or equal to the number of measures or the number of periods, respectively.

IntegralMeasureMatrixSW

This extends `IntegralMeasureMatrix` to add a sliding window for the records. With the base class, the total number of records is limited to the number of periods in the measure matrix. With this class, the number of added records can be greater than the number of periods. If a record is added while all allocated periods are used, the first record is lost and the new one is added. Therefore, the integral can be obtained for the last periods only.

```
package umontreal.iro.lecuyer.stat.mperiods;

public class IntegralMeasureMatrixSW<M extends MeasureMatrix> extends
    IntegralMeasureMatrix<M>
```

Constructor

```
public IntegralMeasureMatrixSW (M mat, int numPeriods)
    Calls super (mat, numPeriods).
```

Methods

```
public int getFirstRealRecord()

    Returns the first value  $p$  for which a recorded value  $\mathbf{f}(t_p)$  is available. If no recorded value is lost, this returns 0.
```

Returns the first value of p for which $\mathbf{f}(t_p)$ is available.

```
public void setFirstRealRecord (int firstRealRecord)

    Sets the index of the first real record to firstRealRecord.
```

Parameter

`firstRealRecord` the index of the first real record.

Throws

`IllegalArgumentException` if `firstRealRecord` is negative.

```
public int getNumRealRecords()

    Returns the total number of times the newRecord() method was called since the last call to IntegralMeasureMatrix.init() plus one. If the returned value exceeds the number of stored records (IntegralMeasureMatrix.getNumStoredRecords()), only the values of  $\mathbf{f}(t)$  for the last IntegralMeasureMatrix.getNumStoredRecords() are accessible; the first values are then lost.
```

Returns the total number of records.

```
public void newRecord()
```

This is the same as in the superclass, but if the number of stored records exceeds the number of real records, the first stored record is discarded.

```
protected int getPeriod()
```

Returns `mpc.getNumRealPeriods()`.

```
public double getSum (int i, int r)
```

Returns $f_i(t_j)$, the measure `i` of the associated measure matrix at the simulation time t_j , j being `r + getFirstRealRecord()`.

Parameters

`i` the measure index.

`r` the record index.

Returns the value of the measure.

Throws

`IndexOutOfBoundsException` if `i` or `r` are out of bounds.

```
public double getMeasure (int i, int p)
```

Returns the measure `i` for period `p`. This corresponds to $f_i(t_{s+p+1}) - f_i(t_{s+p})$ where t_{s+p} is the simulation time of the stored record p , and s is the result of `getFirstRealRecord()`.

Parameters

`i` the index of the measure.

`p` the period of the measure.

Returns the corresponding value.

Throws

`IndexOutOfBoundsException` if `i` or `p` are negative or greater than or equal to the number of measures or the number of periods, respectively.

Package `umontreal.iro.lecuyer.simevents`

Extends the SSJ's discrete-event simulation package to add utility methods and support batch means simulation.

SimTimeMeasureMatrix

This matrix of measures contains a single measure corresponding to the current simulation time.

```
package umontreal.iro.lecuyer.simevents;  
  
public class SimTimeMeasureMatrix implements MeasureMatrix
```

Methods

```
public void setNumMeasures (int nm)  
    Throws an UnsupportedOperationException.  
  
    Throws  
        UnsupportedOperationException if this method is called.  
  
public void setNumPeriods (int np)  
    Throws an UnsupportedOperationException.  
  
    Throws  
        UnsupportedOperationException if this method is called.  
  
public void regroupPeriods (int x)  
    Throws an UnsupportedOperationException.  
  
    Throws  
        UnsupportedOperationException if this method is called.
```

UnusableSimulator

Simulator for which all methods throw an `UnsupportedOperationException`. By setting `Simulator.defaultSimulator` to an instance of this class, one can detect unexpected usage of the static `Sim` class. This can be useful for adapting a program for parallel simulations, because such a program must use an instance of `Simulator` for each parallel replication rather than the static class. An unexpected use of `Sim` may lead to unpredictable results in such cases.

```
package umontreal.iro.lecuyer.simevents;  
  
public class UnusableSimulator extends Simulator
```


References

- [1] O. Z. Akşin, M. Armony, and V. Mehrotra. The modern call center: A multi-disciplinary perspective on operations management research. *Production and Operations Management*, 16(6):665–688, 2007.
- [2] A. N. Avramidis, W. Chan, and P. L’Ecuyer. Staffing multi-skill call centers via search methods and a performance approximation. *IIE Transactions*, 41:483–497, 2009.
- [3] A. N. Avramidis, A. Deslauriers, and P. L’Ecuyer. Modeling daily arrivals to a telephone call center. *Management Science*, 50(7):896–908, 2004.
- [4] A. N. Avramidis and P. L’Ecuyer. Modeling and simulation of call centers. In M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, editors, *Proceedings of the 2005 Winter Simulation Conference*, pages 144–152. IEEE Press, 2005.
- [5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, September 2001.
- [6] J. M. Davenport and R. L. Iman. An iterative algorithm to produce a positive definite correlation matrix from an approximate correlation matrix. Technical report, Sandia National Laboratories, Albuquerque, New Mexico, 1982.
- [7] 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.
- [8] N. Gans, G. Koole, and A. Mandelbaum. Telephone call centers: Tutorial, review, and research prospects. *Manufacturing and Service Operations Management*, 5:79–141, 2003.
- [9] Wolfgang Hoschek. *The Colt Distribution: Open Source Libraries for High Performance Scientific and Technical Computing in Java*. CERN, Geneva, 2004. Available at <http://acs.lbl.gov/software/colt/>.
- [10] N. L. Johnson and S. Kotz. *Distributions in Statistics: Discrete Distributions*. Houghton Mifflin, Boston, 1969.
- [11] G. Jongbloed and G. Koole. Managing uncertainty in call centers using Poisson mixtures. Manuscript, Vrije University, Amsterdam.
- [12] G. Koole, A. Pot, and J. Talim. Routing heuristics for multi-skill call centers. In *Proceedings of the 2003 Winter Simulation Conference*, pages 1813–1816. IEEE Press, 2003.
- [13] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, NY, third edition, 2000.

- [14] P. L'Ecuyer. *SSJ: A Java Library for Stochastic Simulation*, 2004. Software user's guide, available at <http://www.iro.umontreal.ca/~lecuyer>.
- [15] 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.
- [16] J. E. Mosimann. On the compound negative multinomial distribution and correlations among inversely sampled pollen counts. *Biometrika*, 50:47–54, 1963.
- [17] W. Whitt. Engineering solution of a basic call-center model. *Management Science*, 2004. To appear.